LILLIPUT-AE: a New Lightweight Tweakable Block Cipher for Authenticated Encryption with Associated Data

Submission to the NIST Lightweight Cryptography Standardization Process

Alexandre Adomnicai[1], Thierry P. Berger[2], Christophe Clavier[2], Julien Francq[3], Paul Huynh[4], Virginie Lallemand[4], Kévin Le Gouguec[3], Marine Minier[4], Léo Reynaud[2], and Gaël Thomas[5]

[1]Trusted Objects - Europarc de Pichaury, Bât. B8, 1330 rue Guillibert de la Lauzière, 13290 AIX-EN-PROVENCE - France, email: `a.adomnicai@trusted-objects.com`
[2]Université de Limoges - 123 avenue Albert Thomas, 87060 LIMOGES Cedex - France, email: `thierry.berger, christophe.clavier, leo.reynaud@xlim.fr`
[3]Airbus CyberSecurity - ZA Clef Saint-Pierre, 1 Bd Jean Moulin, CS 40001, MetaPole, 78996 ELANCOURT Cedex - France, email: `julien.francq, kevin.legouguec@airbus.com`
[4]Université de Lorraine, CNRS, Inria, LORIA - Campus Scientifique - BP 239, 54506 VANDOEUVRE-LES-NANCY - France, email: `paul.huynh, virginie.lallemand, marine.minier@loria.fr`
[5]DGA Maîtrise de l'information - BP 7, 35998 RENNES CEDEX 9 - France, email: `gael.thomas.87@gmail.com`

Corresponding Submitter's Name: Julien Francq
Email: `julien.francq@airbus.com`
Telephone: (+33) 1 61 38 71 39
Organization: Airbus CyberSecurity
Postal Address: Airbus CyberSecurity - ZA Clef Saint-Pierre, 1 Bd Jean Moulin, CS 40001, MetaPole, 78996 ELANCOURT Cedex - France

Backup Point of contact: Marine Minier
Email: `marine.minier@loria.fr`
Telephone: (+33) 6 87 10 68 58
Organization: Université de Lorraine, CNRS, Inria, LORIA
Postal Address: Université de Lorraine, CNRS, Inria, LORIA - Campus Scientifique - BP 239, 54506 VANDOEUVRE-LES-NANCY - France

website: `https://paclido.fr/lilliput-ae/`

Contents

Chapter 1

Introduction

In this submission to the NIST Lightweight Cryptography Standardization Process, we present a new Authenticated Encryption with Associated Data (AEAD) scheme Lilliput-AE based on the tweakable block cipher Lilliput-TBC which is itself based on the classical block cipher Lilliput presented in [6] with a modified tweakey schedule.

We define two particular authenticated encryption modes: Lilliput-I and Lilliput-II based respectively on the two modes ΘCB3 [38] and SCT-2 used in Deoxys [33] for both. The ΘCB3 mode is a nonce-respecting mode whereas SCT-2 is a nonce-misuse resistant mode.

From those two authenticated encryption modes Lilliput-I and Lilliput-II, we derive several sets of parameters that conform with the NIST Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process. Our primary member is Lilliput-II-128.

As shown in the next chapters, Lilliput-AE is an authenticated encryption scheme that provides full 128-bit, 192-bit or 256-bit security level. It performs well in software and also in hardware. Moreover, the underlying block cipher Lilliput has been extensively studied by the cryptographic community [53, 40, 51] and no weakness has been exhibited for the full version of Lilliput.

We are convinced that extending Lilliput, a well-studied lightweight block cipher, to an 8-bit oriented version and combining it with a mode with good performances and with reinforced security, is a good answer regarding both efficiency and security to the expectations of the NIST standardization process.

Main Features of Lilliput-AE. From our point of view, Lilliput-AE brings many advantages:

- It is based on building schemes (authenticated encryption modes, encryption process) that have been significantly studied by the cryptographic community. Moreover, the security of these blocks has been strengthened by modifiying some parameters (e.g., more secure S-box and tweakey schedule).

- Its primary member is a nonce-misuse resistant mode, which allows an easier management of cryptographic components deployed on the field.

- Its software implementations on 8-bit (e.g., Atmel AVR ATmega128 microcontrollers) and 16-bit (e.g., Texas Instruments MSP430F1611 microcontrollers) platforms are very competitive. In terms of execution time (which relates to power consumption), and for 128-bit keys, Lilliput-AE is comparable to lightweight winners of the CAESAR competition [16], ACORN and Ascon, on 8-bit platforms, and is significantly faster on 16-bit platforms.

- Its hardware implementations on FPGA platforms (e.g., Xilinx Spartan-6) are more compact than ACORN and Ascon. Moreover, straightforward ASIC implementations of Lilliput-AE lead to at most around 5000 Gate Equivalents (GEs) for its maximum parameter sizes. Serial implementations will decrease this figure down to 4000 GEs or 3000 GEs depending on the parameter sizes, which is equivalent to serial implementations of plain AES without authentication mode.

- Some degrees of freedom are given to the implementers of LILLIPUT-AE: for some operations (e.g., in the tweakey schedule), they can trade code size for RAM usage and execution time. Some operations can also be tabulated to accelerate their computation.
- The design facilitates side-channel protection: in particular, the S-box of LILLIPUT-AE has been chosen to optimize its cost in threshold implementations.
- A first fault injection analysis of LILLIPUT-AE shows that faulting 7 rounds or more from the end of the algorithm requires injecting too many faults (say millions) to be practical. A cautious recommendation is then to protect the last 7 rounds of LILLIPUT-AE against fault injection, which leads to a 22% execution time overhead if a straightforward duplication countermeasure is implemented.

Organization of the submission. In Chapter 2, we provide the complete specifications of our submission LILLIPUT-AE including the two considered modes of authenticated encryption with associated data LILLIPUT-I and LILLIPUT-II (Section 2.2) and the tweakable block cipher LILLIPUT-TBC with its particular tweakey schedule (Section 2.3).

In Chapter 3, we detail our design choices: first for the modes (Section 3.1) and second for the tweakable block cipher (Section 3.2). We also perform an extensive security analysis of these two parts in Section 3.3 and in Section 3.4.

In Chapter 4, we give the implementation results we obtain for both software platforms and hardware platforms.

Chapter 2

Specifications

This chapter presents the full specifications of our submission to the NIST Lightweight Cryptography Standardization Process. More precisely, we present our new Authenticated Encryption with Associated Data (AEAD) scheme Lilliput-AE.

After introducing notations and the sets of parameters, we introduce in Section 2.2 the two particular authenticated encryption modes: Lilliput-I based on the nonce-respecting mode ΘCB3 and Lilliput-II based on the nonce-misuse resistant mode SCT-2.

Then, in Section 2.3, we introduce our tweakable block cipher Lilliput-TBC used in both Lilliput-I and Lilliput-II.

Notations. Let us introduce the following notations: K will represent the key of length k bits, P the plaintext of length n bits, T the tweak of length t bits and we denote by $E_K(T, P)$ the ciphering process using the tweakable block cipher E_K^T.

The concatenation operation at binary level is represented by $||$ and $pad10^*$ is the function that applies the 10^* padding on n bits, i.e. $pad10^*(X) = X||1||0^{n-|X|-1}$ when $|X| < n$. For an empty string ϵ, the 10^* padding will not add any bit: $pad10^*(\epsilon) = \epsilon$. The truncation of the word X to the first i bits is given by $\lceil X \rceil_i$, and the truncation to the last i bits by $\lfloor X \rfloor_i$. To emphasize a string X is of length n, we may write it $X_{(n)}$. We denote by $\gg i$ and $\ll i$ respectively the right and left shifts of i bits, and by $\ggg i$ and $\lll i$ the right and left rotations of i bits.

We will also denote by $S^{\gg i}$ and $S^{\ll i}$ the binary matrices of size 8×8 corresponding to a right shift by i bits positions or a left shift by i bits positions respectively. More precisely, and for example,

$$S^{\gg 1} = \begin{pmatrix} 0&0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0 \end{pmatrix} \text{ and } S^{\ll 1} = \begin{pmatrix} 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

The encryption part $\mathcal{E}$ takes as input a variable-length plaintext M (with $m = |M|$ bits), a variable-length associated data A (with $a = |A|$ bits), a fixed-length public nonce N and a k-bit key K (we deliberately used the same letter K to represent the key in the authenticated encryption scheme and the one in the tweakable block cipher, since they always refer to the same object). It outputs an m-bit ciphertext C and a τ-bit tag, denoted $\texttt{tag}$ (with $\tau \in [0, \cdots, n]$) i.e. $(C, \texttt{tag}) = \mathcal{E}_K(N, A, M)$.

The verification/decryption part $\mathcal{D}$ takes as input a variable-length ciphertext C (with $m = |C|$), a τ-bit tag, denoted $\texttt{tag}$ (with $\tau \in [0, \cdots, n]$), a variable-length associated data A (with $a = |A|$), a fixed-length public nonce N and a k-bit key K. It outputs either an error string $\perp$ to signify that the verification has failed, or an m-bit string $M = \mathcal{D}_K(N, A, C, \texttt{tag})$ when the tag is valid.

The maximum message length (in n-bit blocks) is denoted max_l and the maximum number of messages that can be handled with the same key is denoted max_m (the same limitation applies to the associated data material).

2.1 Recommended Parameters

We derive our scheme Lilliput-AE into two authenticated encryption modes: Lilliput-I and Lilliput-II. Lilliput-I is a nonce-respecting mode corresponding with ΘCB3 and Lilliput-II is a nonce-misuse resistant mode corresponding with SCT-2.

The recommended parameter sets for all variants of these modes is given in table 2.1. These parameters have been chosen according to the internal tweakable block cipher Lilliput-TBC.

Name	k	t	n	$\|N\|$	τ
Lilliput-I-128	128	192	128	120	128
Lilliput-I-192	192	192	128	120	128
Lilliput-I-256	256	192	128	120	128
Lilliput-II-128	**128**	**128**	**128**	**120**	**128**
Lilliput-II-192	192	128	128	120	128
Lilliput-II-256	256	128	128	120	128

Table 2.1: Recommended parameter sets for Lilliput-AE. Our primary member Lilliput-II-128 is in bold notation.

For both variants, $max_m = 2^{|N|} = 2^{120}$ bits. However, max_l is dependent on the tweak input and thus differs from one variant to the other:

- in the encryption part of Lilliput-I, the tweak is a concatenation of a 4-bit prefix, the nonce N and the index of the message block, thus $max_l = 2^{t-4-|N|}$ blocks.
- in the encryption part of Lilliput-II, the tweak is a concatenation of a 4-bit prefix and the index of the message block, thus $max_l = 2^{t-4}$ blocks.

As a result, the maximum message length in bytes is 2^{72} bytes for Lilliput-I and 2^{128} bytes for Lilliput-II.

2.2 The Authenticated Encryption Lilliput-AE

In this section, we describe the authenticated encryption modes that are used in our proposal Lilliput-AE. These mode variants are similar to the two modes described in Deoxys [33]:

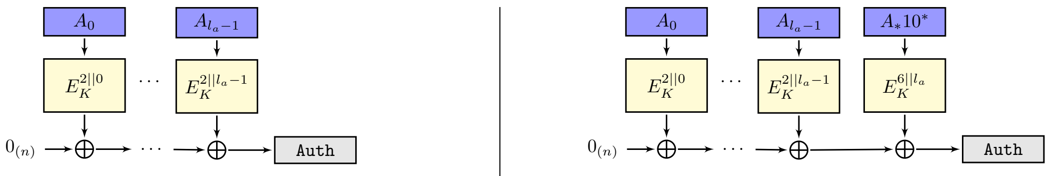
- Lilliput-I (Section 2.2.1): in this nonce-respecting variant, the same nonce N is expected to never be used twice with the same key for encryption. $\mathcal{E}^I$ denotes the encryption part and $\mathcal{D}^I$ the verification/decryption part.
- Lilliput-II (Section 2.2.2): in this variant, a nonce N may be reused with the same key for encryption. $\mathcal{E}^{II}$ denotes the encryption part and $\mathcal{D}^{II}$ the verification/decryption part.

As stated previously, 4-bit prefixes are used for the tweak input to separate the various types of encryption/authentication blocks, akin to what has been done in Deoxys [33].

2.2.1 Nonce-Respecting Mode: ΘCB3

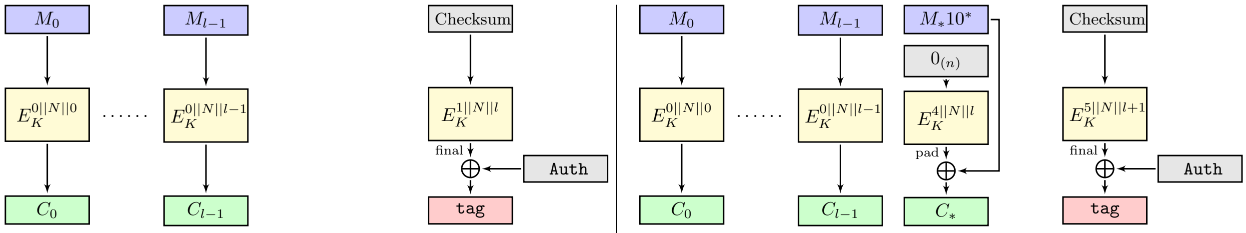
This scheme follows the ΘCB3 framework [38] and therefore directly benefits from this framework's proof of security regarding authentication and privacy. In this mode, the tweak length is 192 bits. The encryption algorithm $\mathcal{E}^I$ is given in Algorithm 1 while the verification/decryption algorithm $\mathcal{D}^I$ is given in Algorithm 2.

If the length of the associated data is not a multiple of the block size, the final block is padded with the 10^* padding, as depicted in Figure 2.1. The same applies for the message and the ciphertext as shown in Figures 2.2 and 2.3.



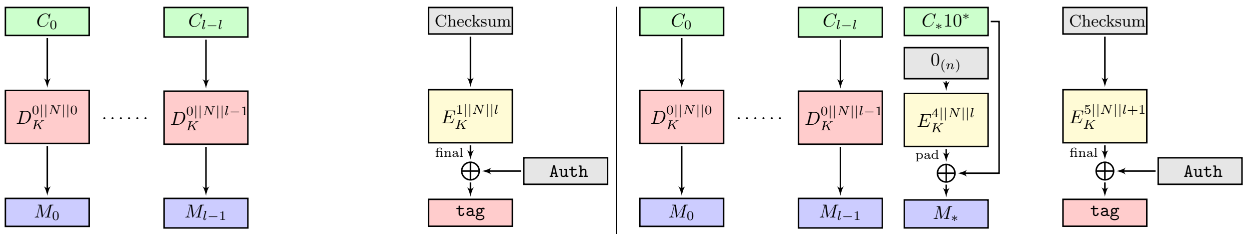
(a) Without padding. (b) With padding.

Figure 2.1: Handling of the associated data in the nonce-respecting mode.



(a) Without padding. (b) With padding.

Figure 2.2: Message processing for the nonce-respecting mode.



(a) Without padding. (b) With padding.

Figure 2.3: Ciphertext processing for the nonce-respecting mode.

Algorithm 1: The encryption algorithm $\mathcal{E}^{\mathtt{I}}_K(N, A, M)$.
In the tweak inputs, the value N is encoded on 120 bits, the integer values j and l are encoded on 68 bits, while the integer values i and l_a are encoded on 188 bits.

/* *Associated data* */
$A_0||\cdots||A_{l_a-1}||A_* \leftarrow A$ where each $|A_i| = n$ and $|A_*| < n$
$\text{Auth} \leftarrow 0_{(n)}$
for $i = 0$ **to** $l_a - 1$ **do**
    $\text{Auth} \leftarrow \text{Auth} \oplus E_K(0010||i, A_i)$
end
if $A_* \neq \epsilon$ **then**
    $\text{Auth} \leftarrow \text{Auth} \oplus E_K(0110||l_a, pad10^*(A_*))$
end

/* *Message* */
$M_0||\cdots||M_{l-1}||M_* \leftarrow M$ where each $|M_j| = n$ and $|M_*| < n$
$\text{Checksum} \leftarrow 0_{(n)}$
for $j = 0$ **to** $l - 1$ **do**
    $\text{Checksum} \leftarrow \text{Checksum} \oplus M_j$
    $C_j \leftarrow E_K(0000||N||j, M_j)$
end
if $M_* = \epsilon$ **then**
    $\text{Final} \leftarrow E_K(0001||N||l, \text{Checksum})$
    $C_* \leftarrow \epsilon$
else
    $\text{Checksum} \leftarrow \text{Checksum} \oplus pad10^*(M_*)$
    $\text{Pad} \leftarrow E_K(0100||N||l, 0_{(n)})$
    $C_* \leftarrow M_* \oplus \lceil \text{Pad} \rceil_{|M_*|}$
    $\text{Final} \leftarrow E_K(0101||N||l+1, \text{Checksum})$
end

/* *Tag generation* */
$\mathtt{tag} \leftarrow \text{Final} \oplus \text{Auth}$
return $(C_0||\cdots||C_{l-1}||C_*, \mathtt{tag})$

Algorithm 2: The verification/decryption algorithm $\mathcal{D}_K^{\mathtt{I}}(N, A, C, \mathtt{tag})$.
In the tweak inputs, the value N is encoded on 120 bits, the integer values j and l are encoded on 68 bits, while the integer values i and l_a are encoded on 188 bits.

/* *Associated data* */
$A_0||\cdots||A_{l_a-1}||A_* \leftarrow A$ where each $|A_i| = n$ and $|A_*| < n$
Auth $\leftarrow 0_{(n)}$
for $i = 0$ **to** $l_a - 1$ **do**
  Auth $\leftarrow$ Auth $\oplus E_K(0010||i, A_i)$
end
if $A_* \neq \epsilon$ **then**
  Auth $\leftarrow$ Auth $\oplus E_K(0110||l_a, pad10^*(A_*))$
end

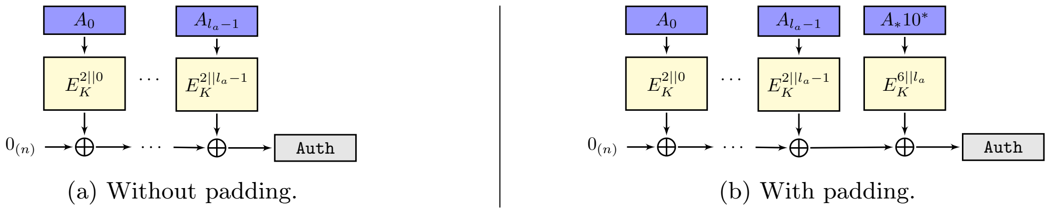
/* *Ciphertext* */
$C_0||\cdots||C_{l-1}||C_* \leftarrow C$ where each $|C_j| = n$ and $|C_*| < n$
Checksum $\leftarrow 0_{(n)}$
for $j = 0$ **to** $l - 1$ **do**
  $M_j \leftarrow D_K(0000||N||j, C_j)$
  Checksum $\leftarrow$ Checksum $\oplus M_j$
end
if $C_* = \epsilon$ **then**
  Final $\leftarrow E_K(0001||N||l,$ Checksum$)$
  $M_* \leftarrow \epsilon$
else
  Pad $\leftarrow E_K(0100||N||l, 0_{(n)})$
  $M_* \leftarrow C_* \oplus \lceil \text{Pad} \rceil_{|C_*|}$
  Checksum $\leftarrow$ Checksum $\oplus pad10^*(M_*)$
  Final $\leftarrow E_K(0101||N||l+1,$ Checksum$)$
end

/* *Tag generation* */
$\mathtt{tag}' \leftarrow$ Final $\oplus$ Auth
if $\mathtt{tag}' = \mathtt{tag}$ **then**
  return $(M_0||\cdots||M_{l-1}||M_*)$
else
  return $\perp$
end

2.2.2 Nonce-Misuse Resistant Mode

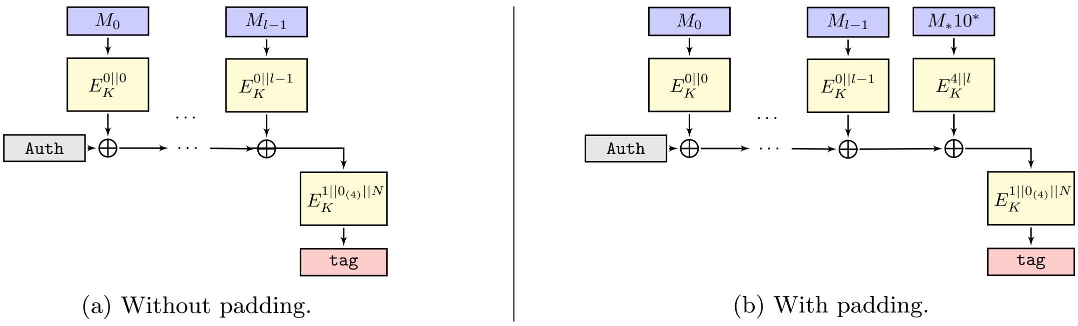
This scheme is the variant of SCT introduced in Deoxys [33]: SCT-2. In this mode, the tweak length is 128 bits while the size of the nonce N remains unchanged and is 120 bits. The encryption algorithm $\mathcal{E}^{II}$ is given in Algorithm 3 while the verification/decryption algorithm $\mathcal{D}^{II}$ is given in Algorithm 4.

The associated data is processed as in the previous variant, as depicted in Figure 2.4. The processing of the message is shown in Figures 2.5 and 2.6 and decryption is shown in Figure 2.7.



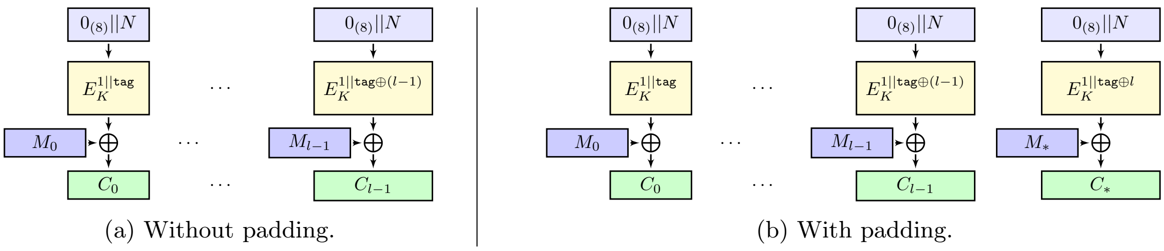
(a) Without padding. (b) With padding.

Figure 2.4: Handling of the associated data in the nonce-misuse resistant mode.



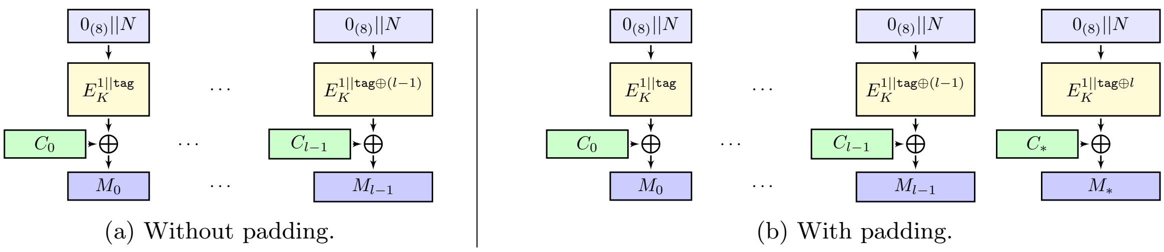
(a) Without padding. (b) With padding.

Figure 2.5: Message processing in the authentication part of the nonce-misuse resistant mode.



(a) Without padding. (b) With padding.

Figure 2.6: Message processing in the encryption part of the nonce-misuse resistant mode.



(a) Without padding. (b) With padding.

Figure 2.7: Ciphertext processing in the decryption part of the nonce-misuse resistant mode.

Algorithm 3: The encryption algorithm $\mathcal{E}_K^{\mathrm{II}}(N, A, M)$.
In the tweak inputs, the integer values i, j, l and l_a are encoded on 124 bits. Moreover, $\mathtt{tag} \oplus j$ values are encoded on 127 bits (the most significant bit is truncated since $|\mathtt{tag}| = \tau$).

/* *Associated data* */
$A_0 || \cdots || A_{l_a - 1} || A_* \leftarrow A$ where each $|A_i| = n$ and $|A_*| < n$
Auth $\leftarrow 0_{(n)}$
for $i = 0$ **to** $l_a - 1$ **do**
  Auth $\leftarrow$ Auth $\oplus E_K(0010 || i, A_i)$
end
if $A_* \neq \epsilon$ **then**
  Auth $\leftarrow$ Auth $\oplus E_K(0110 || l_a, pad10^*(A_*))$
end

/* *Message authentication and tag generation* */
$M_0 || \cdots || M_{l-1} || M_* \leftarrow M$ where each $|M_j| = n$ and $|M_*| < n$
$\mathtt{tag} \leftarrow$ Auth
for $j = 0$ **to** $l - 1$ **do**
  $\mathtt{tag} \leftarrow \mathtt{tag} \oplus E_K(0000 || j, M_j)$
end
if $M_* \neq \epsilon$ **then**
  $\mathtt{tag} \leftarrow \mathtt{tag} \oplus E_K(0100 || l, pad10^*(M_*))$
end
$\mathtt{tag} \leftarrow E_K(0001 || 0^4 || N, \mathtt{tag})$

/* *Message encryption* */
for $j = 0$ **to** $l - 1$ **do**
  $C_j \leftarrow M_j \oplus E_K(1 || \mathtt{tag} \oplus j, 0^8 || N)$
end
if $M_* \neq \epsilon$ **then**
  $C_* \leftarrow M_* \oplus \lceil E_K(1 || \mathtt{tag} \oplus l, 0^8 || N) \rceil_{|M_*|}$
else
  $C_* \leftarrow \epsilon$
end

return $(C_0 || \cdots || C_{l-1} || C_*, \mathtt{tag})$

Algorithm 4: The verification/decryption algorithm $\mathcal{D}_K^{\mathrm{II}}(N, A, C, \texttt{tag})$.
In the tweak inputs, the integer values i, j, l and l_a are encoded on 124 bits. Moreover, $\texttt{tag} \oplus j$ values are encoded on 127 bits (the most significant bit is truncated since $|\texttt{tag}| = \tau$).

/* *Message decryption* */
$C_0||\cdots||C_{l-1}||C_* \leftarrow C$ where each $|C_j| = n$ and $|C_*| < n$
for $j = 0$ **to** $l - 1$ **do**
  $M_j \leftarrow C_j \oplus E_K(1||\texttt{tag} \oplus j, 0^8||N)$
end
if $C_* \neq \epsilon$ **then**
  $M_* \leftarrow C_* \oplus \lceil E_K(1||\texttt{tag} \oplus l, 0^8||N) \rceil_{|C_*|}$
else
  $M_* \leftarrow \epsilon$
end

/* *Associated data* */
$A_0||\cdots||A_{l_a-1}||A_* \leftarrow A$ where each $|A_i| = n$ and $|A_*| < n$
Auth $\leftarrow 0_{(n)}$
for $i = 0$ **to** $l_a - 1$ **do**
  Auth $\leftarrow$ Auth $\oplus E_K(0010||i, A_i)$
end
if $A_* \neq \epsilon$ **then**
  Auth $\leftarrow$ Auth $\oplus E_K(0110||l_a, pad10^*(A_*))$
end

/* *Message authentication and tag generation* */
$M_0||\cdots||M_{l-1}||M_* \leftarrow M$ where each $|M_j| = n$ and $|M_*| < n$
$\texttt{tag}' \leftarrow$ Auth
for $j = 0$ **to** $l - 1$ **do**
  $\texttt{tag}' \leftarrow \texttt{tag}' \oplus E_K(0000||j, M_j)$
end
if $M_* \neq \epsilon$ **then**
  $\texttt{tag}' \leftarrow \texttt{tag}' \oplus E_K(0100||l, pad10^*(M_*))$
end
$\texttt{tag}' \leftarrow E_K(0001||0^4||N, \texttt{tag}')$

/* *Tag verification* */
if $\texttt{tag}' = \texttt{tag}$ **then**
  return $(M_0||\cdots||M_{l-1}||M_*)$
else
  return $\perp$
end

2.3 The Tweakable Block Cipher Lilliput-TBC

In this section we present our dedicated lightweight Tweakable Block Cipher Lilliput-TBC that is based on the EGFN [8] described in Fig. 2.8.

Lilliput-TBC is composed of 6 variants depending on the key lengths (possible key lengths are equal to 128, 192 and 256 bits) and on the tweak lengths (possible tweak lengths are equal to 128 or 192 bits). The different parameters for those variants are specified in Table 2.2. Lilliput-TBC-I for the three possible key lengths and a tweak length equal to 192 bits will be used in the mode Lilliput-I and Lilliput-TBC-II for the three possible key lengths and a tweak length equal to 128 bits will be used in the mode Lilliput-II.

Name	k	t	Nb of rounds r
Lilliput-TBC-I-128	128	192	32
Lilliput-TBC-I-192	192	192	36
Lilliput-TBC-I-256	256	192	42
Lilliput-TBC-II-128	128	128	32
Lilliput-TBC-II-192	192	128	36
Lilliput-TBC-II-256	256	128	42

Table 2.2: Recommended parameter sets for Lilliput-TBC.

2.3.1 Encryption Process

Lilliput-TBC is a 128-bit tweakable block cipher with key sizes of 128, 192 or 256 bits and tweak sizes of 128 or 192 bits. The whole encryption process is depicted in Fig. 2.9. As previously explained, Lilliput-TBC uses an Extended Generalized Feistel Network (EGFN) with a 128-bit state and a round function acting at byte level. The state X is seen as 16 bytes, denoted $X_{15}, \cdots, X_0$. In its 128-bit key version, the cipher is composed of $r = 32$ rounds, *i.e.* 32 repetitions of a single EGFN called `OneRoundEGFN`, depicted in Fig. 2.8. Each F_j for j from 0 to 7 is defined as $F_j = S(X_j \oplus RTK^i_j)$ where S is an S-box that acts at byte level and RTK^i_j is the byte of position j of the 64-bit subtweakey RTK^i of round i. The 32 64-bit subtweakeys RTK^i are generated from the master key and the tweak using the tweakey schedule.

In more details, the round function denoted `OneRoundEGFN` in Fig. 2.9 is composed of a layer of non linear components called `NonLinearLayer` for confusion; a new layer called `LinearLayer` in [8] that represent a linear layer made of linear components applied in a Feistel way; and a block-wise permutation called `PermutationLayer` for diffusion. All three layers act at byte level on the EGFN state X and together constitute one iteration of the EGFN, as shown in Fig. 2.8.

Note that with this new layer `LinearLayer`, it is possible to shuffle blocks better than what was possible using the block-wise permutation only of a classical Feistel scheme, while preserving the self-invertibility of the scheme.

Note that the last round skips the `PermutationLayer` for involution reasons.

For the 192-bit and 256-bit key versions, the number of rounds r is 36 and 42 respectively.

Overview of the EGFN round function

The particular EGFN we use in Lilliput-TBC with $k = 16$ blocks is depicted in Fig. 2.8.

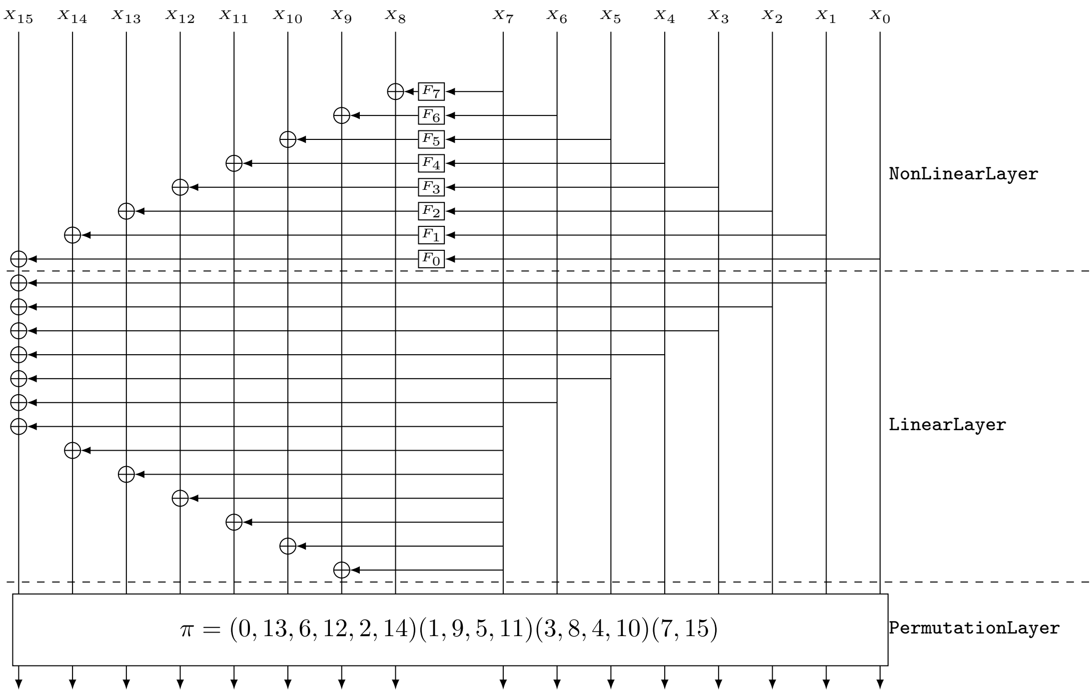


Figure 2.8: The EGFN used in LILLIPUT-TBC that reaches full diffusion in $d = 4$ rounds. The permutation π is given as a product of cycles and can also be found in Table 2.3.

In more details, `OneRoundEGFN` is composed of:

- `NonLinearLayer`: It is the non-linear part of the EGFN and is made of 8 parallel updates of the EGFN state. Each F_j for j from 0 to 7 is defined as $F_j = S(X_j \oplus RTK_j^i)$ where S is an S-box that acts at byte level given in Table 2.4 and RTK_j^i is the byte of position j of the 64-bit subtweakey RTK^i of round i.
- `LinearLayer`: It aims at providing quick diffusion between bytes and consists in xoring some bytes to some other bytes. More precisely, as depicted in Fig 2.8, blocks X_1 to X_6 are xored to block X_{15}, and block X_7 is xored to blocks X_9 to X_{15}.
- `PermutationLayer`: It consists in applying the permutation π given in Table 2.3 to the bytes.

The permutation π used in `PermutationLayer`

The permutation π is given in Table 2.3. It has been chosen to maximize the number of active S-boxes on 18, 19 and 20 rounds as it will be shown in Section 3.4. For each round $i \in \{0, \cdots, r-1\}$, let us denote Y^i the output at round i after the transformations `NonLinearLayer` and `LinearLayer` with $Y^i = (Y_{15}^i, \cdots, Y_0^i)$ its byte representation, i.e. $Y^i = (Y_{15}^i, \cdots, Y_0^i) =$ `LinearLayer`(`NonLinearLayer`(X^i)). Then, the `PermutationLayer` is applied on Y^i in the following way:

$$\forall i \in \{1, \cdots, r-2\}, \forall j\{0, \cdots, 15\} \in \quad X_{\pi(j)}^i = Y_j^{i-1}.$$

Table 2.3: Block permutation π used in encryption mode and its inverse π^{-1} used in decryption mode.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi(i)$	13	9	14	8	10	11	12	15	4	5	3	1	2	6	0	7
$\pi^{-1}(i)$	14	11	12	10	8	9	13	15	3	1	4	5	6	0	2	7

The S-box S used in `NonLinearLayer`

The S-box S used in `NonLinearLayer` is the 8-bit S-box given in Table 2.4. The properties of this S-box will be described in Section 3.2.3 of the Chapter 3.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	20	00	B2	85	3B	35	A6	A4	30	E4	6A	2C	FF	59	E2	0E
10	F8	1E	7A	80	15	BD	3E	B1	E8	F3	A2	C2	DA	51	2A	10
20	21	01	23	78	5C	24	27	B5	37	C7	2B	1F	AE	0A	77	5F
30	6F	09	9D	81	04	5A	29	DC	39	9C	05	57	97	74	79	17
40	44	C6	E6	E9	DD	41	F2	8A	54	CA	6E	4A	E1	AD	B6	88
50	1C	98	7E	CE	63	49	3A	5D	0C	EF	F6	34	56	25	2E	D6
60	67	75	55	76	B8	D2	61	D9	71	8B	CD	0B	72	6C	31	4B
70	69	FD	7B	6D	60	3C	2F	62	3F	22	73	13	C9	82	7F	53
80	32	12	A0	7C	02	87	84	86	93	4E	68	46	8D	C3	DB	EC
90	9B	B7	89	92	A7	BE	3D	D8	EA	50	91	F1	33	38	E0	A9
A0	A3	83	A1	1B	CF	06	95	07	9E	ED	B9	F5	4C	C0	F4	2D
B0	16	FA	B4	03	26	B3	90	4F	AB	65	FC	FE	14	F7	E3	94
C0	EE	AC	8C	1A	DE	CB	28	40	7D	C8	C4	48	6B	DF	A5	52
D0	E5	FB	D7	64	F9	F0	D3	5E	66	96	8F	1D	45	36	CC	C5
E0	4D	9F	BF	0F	D1	08	EB	43	42	19	E7	99	A8	8E	58	C1
F0	9A	D4	18	47	AA	AF	BC	5B	D5	11	D0	B0	70	BB	0D	BA

Table 2.4: The S-box in hexadecimal notation. The column indicates the least significant nibble and the row indicates the most significant nibble of the S-box input.

Overall encryption process

Fig. 2.9 gives an overview of the complete encryption process of LILLIPUT-TBC for all its variants.

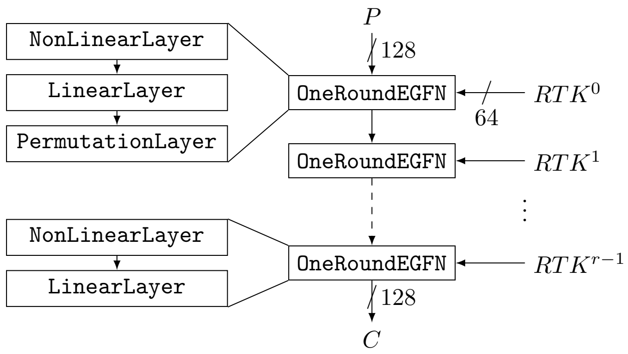


Figure 2.9: LILLIPUT-TBC Encryption process.

2.3.2 Decryption Process

As LILLIPUT-TBC is a Feistel network, decryption is quite analogous to encryption but uses the inverse block permutation π^{-1} given in Table 2.3 and the subkeys in the reverse order. Note that the tweakey process could be inverted at low cost.

2.3.3 Tweakey Schedule

An adapted version of the TWEAKEY framework [32] was used as a building block for the scheduling of the key and the tweak. More specifically, we used a variant of the STK construction, where the key and the tweak inputs are handled almost the same way. The proposed version is depicted in Fig. 2.10.

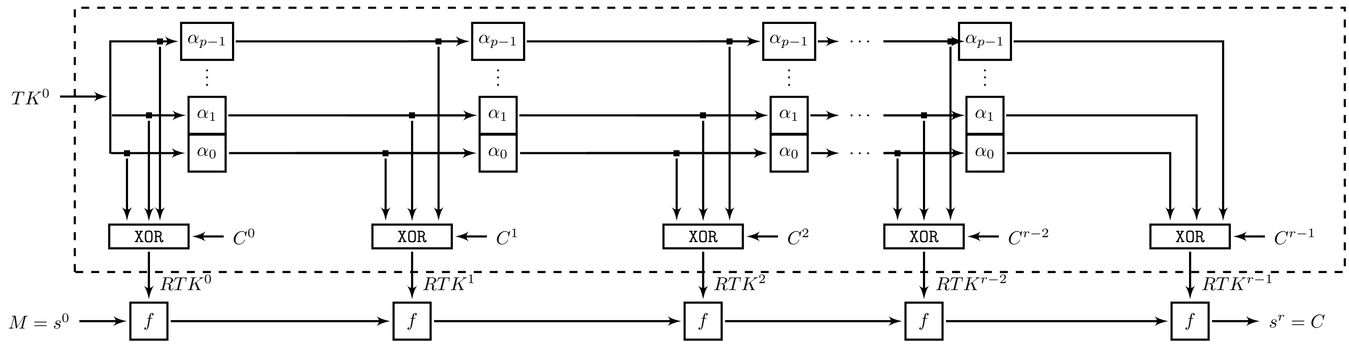


Figure 2.10: The tweakey schedule. f represents the round function `OneRoundEGFN`.

The tweakey schedule produces the $r = 32$ (36 or 42 respectively) 64-bit subtweakeys RTK^0 to RTK^{r-1} from the 128-bit (192 or 256 respectively) master key K and the tweak T that is 128 bits long when Lilliput-TBC-II is used and 192 bits long tweak when Lilliput-TBC-I is used.

As done in the STK construction, at each round $i \in \{0, \cdots, r-1\}$, the inner state TK^i is divided into $p = (t+k)/64$ lanes that we denote TK^i_j, $j \in \{0, \cdots, p-1\}$, where k is the key length and t is the tweak length. The values of p are shown in Table 2.5, depending on which version of Lilliput-TBC is used.

Name	k	t	p	r
Lilliput-TBC-I-128	128	192	5	32
Lilliput-TBC-I-192	192	192	6	36
Lilliput-TBC-I-256	256	192	7	42
Lilliput-TBC-II-128	128	128	4	32
Lilliput-TBC-II-192	192	128	5	36
Lilliput-TBC-II-256	256	128	6	42

Table 2.5: Recommended parameter sets for Lilliput-TBC and associated number of tweakey lanes.

TK^0 is initialized with the concatenation of the tweak T and the master key K. The first 2 (or 3) lanes are thus dedicated to the 128-bit (or 192-bit) tweak. The key is then stored in the following 2, 3 or 4 lanes, depending on its size.

For each round i, the 8-byte subtweakey word that is produced is denoted RTK^i:

$$\forall i \in \{0, \cdots, r-1\}, \quad RTK^i = RTK^i_7 || RTK^i_6 || RTK^i_5 || RTK^i_4 || RTK^i_3 || RTK^i_2 || RTK^i_1 || RTK^i_0,$$

where RTK^i_j is the byte that is xored to block X_j then used as an input of the nonlinear function F_j in the Lilliput-TBC round function of the encryption process.

The subtweakey word is obtained by xoring all p TK^i_j lanes and a round-dependent constant denoted C^i together. In our proposal, the round constant C^i is simply the round number i:

$$\forall i \in \{0, \cdots, r-1\}, \quad RTK^i = \bigoplus_{j=0}^{p-1} TK^i_j \oplus i.$$

To update the tweakey, at each round $i \in \{1, \cdots, r-1\}$, each 64-bit lane TK^i_j is multiplied by a nonzero coefficient denoted α_j, with $j \in \{0, \cdots, p-1\}$. The first coefficient α_0 is set to 1 and corresponds to the identity function. The other coefficients were carefully chosen such that in r consecutive rounds, at most $(p-1)$ cancellations occur as will be shown in Section 3.2. Next, we describe how to generate the sequences induced by coefficients α_j $(j = 1, \cdots, 6)$.

Sequences

The α-multiplications are computed using word-ring-LFSRs [7]. The sequences constructed as α-multiplications for the tweakey on $GF(2^{64})$ using word-ring-LFSRs are the following ones: consider first a 64-bit lane in byte notation as $x = (x_7, \cdots, x_0)$ where x_7 is the most significant byte and x_0 is the least significant one. In binary notations, we obtain the following vector of 64 bits: $x = (x^b_{63}, \cdots, x^b_0)$. Thus, we have, $\alpha_0 = I$, where I is the 64×64 identity matrix, $\alpha_1 = M$, $\alpha_2 = M^2$, $\alpha_3 = M^3$, $\alpha_4 = M_R$, $\alpha_5 = M_R^2$ and $\alpha_6 = M_R^3$.

Then the sequence generated by α_1 is produced using the ring-LFSR represented at byte level word by the following matrix:

$$M = \begin{pmatrix} 0 & Id & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & Id & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & S^{\ll 3} & Id & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & S^{\gg 3} & Id & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & Id & 0 & 0 \\ 0 & S^{\ll 2} & 0 & 0 & 0 & 0 & Id & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & Id \\ Id & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

where Id is the 8×8 identity matrix. The primitive polynomial associated with this matrix is computed as $Det(I - M \cdot X)$ at binary level, which gives: $x^{64} + x^{58} + x^{42} + x^{40} + x^{35} + x^{34} + x^{29} + x^{26} + x^{24} + x^{23} + x^{19} + x^{10} + 1$. The multiplication by α_1 is then generated as $(y_7, \cdots, y_0)^t = M \cdot (x_7, \cdots, x_0)^t$. Thus, we have: $(y_7, \cdots, y_0)^t = (x_6, x_5, x_4 \oplus x_5 \ll 3, x_3 \oplus x_4 \gg 3, x_2, x_1 \oplus x_6 \ll 2, x_0, x_7)^t$.

$$M^2 = \begin{pmatrix} 0 & 0 & Id & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & S^{\ll 3} & Id & 0 & 0 & 0 & 0 \\ 0 & 0 & S^{\ll 6} & M_1 & Id & 0 & 0 & 0 \\ 0 & 0 & 0 & S^{\gg 6} & S^{\gg 3} & Id & 0 & 0 \\ 0 & S^{\ll 2} & 0 & 0 & 0 & 0 & Id & 0 \\ 0 & 0 & S^{\ll 2} & 0 & 0 & 0 & 0 & Id \\ Id & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & Id & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

with M_1 equal to the binary 8×8 following matrix:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

and

$$M^3 = \begin{pmatrix} 0 & 0 & S^{\ll 3} & Id & 0 & 0 & 0 & 0 \\ 0 & 0 & S^{\ll 6} & M_1 & Id & 0 & 0 & 0 \\ 0 & 0 & 0 & M_2 & M_1 & Id & 0 & 0 \\ 0 & S^{\ll 2} & 0 & 0 & S^{\gg 6} & S^{\gg 3} & Id & 0 \\ 0 & 0 & S^{\ll 2} & 0 & 0 & 0 & 0 & Id \\ Id & 0 & S^{\ll 5} & S^{\ll 2} & 0 & 0 & 0 & 0 \\ 0 & Id & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & Id & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

with M_2 a binary matrix of size 8×8 equal to

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

To generate the three following sequences, we use the following matrices using the reciprocal primitive polynomial of $x^{64} + x^{58} + x^{42} + x^{40} + x^{35} + x^{34} + x^{29} + x^{26} + x^{24} + x^{23} + x^{19} + x^{10} + 1$ equal to $x^{64} + x^{54} + x^{45} + x^{41} + x^{40} + x^{38} + x^{35} + x^{30} + x^{29} + x^{24} + x^{22} + x^{6} + 1$.

The outputs are then computed in the reverse order using the relation $(y_0, \cdots, y_7)^t = M_R \cdot (x_0, \cdots, x_7)^t$. Note that the associated binary words are also written in the opposite way compared with the computations performed for M, M^2 and M^3. It means that, in this case, at binary level, we have $x = (x_0^b, \cdots, x_{63}^b)$ and $x_i = (x_{8\cdot i+0}^b, \cdots, x_{8\cdot i+7}^b)$.

Thus, we have: $(y_0, \cdots, y_7)^t = (x_1, x_2, x_3 \oplus x_4 \ll 3, x_4, x_5 \oplus x_6 \gg 3, x_6 \oplus x_3 \gg 2, x_7, x_0)^t$.

$$M_R = \begin{pmatrix} 0 & Id & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & Id & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & Id & S^{\ll 3} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & Id & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & Id & S^{\gg 3} & 0 \\ 0 & 0 & 0 & S^{\gg 2} & 0 & 0 & Id & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & Id \\ Id & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$M_R^2 = \begin{pmatrix} 0 & 0 & Id & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & Id & S^{\ll 3} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & Id & S^{\ll 3} & M_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & Id & S^{\gg 3} & 0 \\ 0 & 0 & 0 & S^{\gg 2} & 0 & 0 & Id & S^{\gg 3} \\ 0 & 0 & 0 & 0 & S^{\gg 2} & 0 & 0 & Id \\ Id & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & Id & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

with M_3 a binary matrix of size 8×8 equal to

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and

$$M_R^3 = \begin{pmatrix} 0 & 0 & 0 & Id & S^{\ll 3} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & Id & S^{\ll 3} & M_3 & 0 \\ 0 & 0 & 0 & M_4 & 0 & Id & M_1 & M_3 \\ 0 & 0 & 0 & S^{\gg 2} & 0 & 0 & Id & S^{\gg 3} \\ S^{\gg 3} & 0 & 0 & 0 & S^{\gg 2} & 0 & 0 & Id \\ Id & 0 & 0 & 0 & 0 & S^{\gg 2} & S^{\gg 5} & 0 \\ 0 & Id & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & Id & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

with M_4 is a binary matrix of size 8×8 equal to

$$M_4 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The periods of each sequence given by the word-ring-LFSRs produced by the previous matrices are respectively: $2^{64} - 1$ for the sequences produced using M, M^2, M_R, M_R^2 and $\frac{2^{64}-1}{3}$ for the sequences produced using M^3 and M_R^3.

Chapter 3

Design Rationale and Security Analysis

In this chapter, we will detail the design choices we made for LILLIPUT-TBC and we provide a complete security analysis regarding a wide variety of attacks.

3.1 Design Rationale of the Modes of Operation

3.1.1 ΘCB3

The OCB mode (Offset Codebook Mode) was designed by Phillip Rogaway, who took inspiration from Charanjit Jutla's IAPM (Integrity Aware Parallelizable Mode) [35]. The original authenticated-encryption scheme has then been refined several times, leading to three named versions: OCB1 [49], OCB2 [48] and OCB3 [39]. The main change introduced with OCB2 is the possibility to handle Associated Data (AD), while the modifications made in OCB3 are rather minor (mostly a change in the way offsets are incremented). The OCB mode has many advantages, starting with the fact that it is parallelizable and only requires one block cipher invocation per message block, in contrary to schemes like GCM. In [39], Krovetz and Rogaway also introduced a tweakable block cipher generalization of OCB3 denoted ΘCB3, which is at the source of the mode used for our candidate LILLIPUT-I.

OCB is covered by the United States Patent No. 7,949,129, United States Patent No. 8,321,675, United States Patent No. 7,046,802 and United States Patent No. 7,200,22. Still, it is unclear if ΘCB3 is also covered by patents. This lack of clarity is part of the reason why we selected LILLIPUT-II to be our primary member.

Under the assumption that the underlying (tweakable) block cipher is secure as a strong-PRP (PseudoRandom Permutation), OCB is provably secure and achieves confidentiality and authenticity. Confidentiality means that an adversary cannot make the distinction between OCB outputs and random bits, while authenticity (of ciphertexts) means that she cannot produce a valid nonce-ciphertext pair (different from the ones she previously obtained). Note that the various variants of OCB are not designed to resist to nonce reuse nor to enjoy beyond-birthday-bound security.

3.1.2 SCT-2

The Synthetic Counter in Tweak mode (SCT) was first devised at Crypto 2016 by Thomas Peyrin and Yannick Seurin [44]. Few months later, the mode was slightly modified by the same authors associated with Jérémy Jean and Ivica Nikolic to be used as a mode for one of the member of the family of authenticated ciphers Deoxys v1. 41 [33], their submission to CAESAR (*Competition for Authenticated Encryption: Security, Applicability, and Robustness*). The rearranged mode was named SCT-2, and the corresponding authenticated cipher was coined Deoxys-II.

The difference between SCT and SCT-2 only lies in the way the tag is produced (the encryption part is similar), a change that was done *"in order to provide graceful degradation of security for authentication with the maximal number of repetitions of nonce"* [33].

3.2 Design Rationale of Lilliput-TBC

When designing Lilliput-TBC from the block cipher Lilliput, our overall goal was to maximize diffusion between nibbles or bytes while keeping reasonable implementation performances. This diffusion could be measured using the notion of *full diffusion delay* of [8]. It will be denoted by d and corresponds to the minimum number of rounds needed for all output bytes or nibbles to depend on all input bytes or nibbles. It is closely related to some structural attacks such as impossible differentials or integral attacks, as shown in [57, 8].

We decided to use the EGFN inferred in Lilliput [6] to reach this purpose because the full diffusion delay of the EGFN of Lilliput is equal to $d = 4$ which is the best diffusion delay obtained for a Feistel-like scheme.

We chose a 128-bit state as it is consistent with the NIST requirements. We split that state into 16 bytes so that the block size matches the S-box size, *i.e.* the non-linear layer is made only of 8 parallel calls to an 8-bit S-box and 8 subkey additions. As said before, the π permutation has been chosen to maximize the number of active S-boxes on 18, 19 and 20 rounds (see Section 3.4 and Table 3.2 for more details).

From those results and the security analysis performed in Section 3.4 and summed up in Table 3.5, we also deduced the number of rounds of each instance of Lilliput-I and of Lilliput-II equal to 32, 36 or 42 rounds.

3.2.1 The EGFN Structure

As done in [57, 8], we analyze here the security of our underlying `EGFN` scheme regarding the pseudorandomness of the scheme. Note that the pseudorandomness bounds obtained are generic and essentially depend on the d value. We thus introduce the pseudo-random-permutation advantage (`prp`-advantage) and the strong-pseudo-random-permutation advantage (`sprp`-advantage) of an adversary. For this purpose, we introduce the two advantage notations as:

$$\mathrm{Adv}_{\mathsf{C}}^{\mathsf{prp}}(q) = \max_{\mathsf{A}:q\text{-}\mathsf{CPA}} \left|\Pr[\mathsf{A}^{\mathsf{C}} = 1] - \Pr[\mathsf{A}^{\mathsf{P}_n} = 1]\right| \tag{3.1}$$

$$\mathrm{Adv}_{\mathsf{C}}^{\mathsf{sprp}}(q) = \max_{\mathsf{A}:q\text{-}\mathsf{CCA}} \left|\Pr[\mathsf{A}^{\mathsf{C},\mathsf{C}^{-1}} = 1] - \Pr[\mathsf{A}^{\mathsf{P}_n,\mathsf{P}_n^{-1}} = 1]\right| \tag{3.2}$$

where C is the encryption function of an n-bit block cipher composed of Uniform Random Functions (URFs) as internal modules whereas C^{-1} is its inverse; P_n is an n-bit Uniform Random Permutation (URP) uniformly distributed among all the n-bit permutations; P_n^{-1} is its inverse. The adversary, A, tries to distinguish C from P_n using q queries in a CPA (Chosen Plaintext Attack) and tries to distinguish, always using q queries, $(\mathsf{C}, \mathsf{C}^{-1})$ from $(\mathsf{P}_n, \mathsf{P}_n^{-1})$ in a CCA (Chosen Ciphertext Attack). The notation means that the final guess of the adversary A is either 0 if A thinks that the computations are done using P_n, or 1 if A thinks that the computations are done using C. The maximums of Equations (3.1,3.2) are taken over all possible adversaries A with q queries and an unbounded computational power.

To prove the bounds of our scheme in those models, we recall the result proved in [6]: let $\Phi_{kc,r}$ denote our k-block scheme acting on c-bit blocks with $n = kc$, using r rounds and with diffusion delay d. We then have (the proof can be found in [6]):

Theorem 1 *Given the r-round EGFN $\Phi_{kc,r}$ with k branches acting on c-bit blocks with a diffusion delay d where all c-bit round functions are independent URFs. Then we have:*

$$\mathrm{Adv}_{\Phi_{kc,d+2}}^{\mathsf{prp}}(q) \leq \frac{kd}{2^c} q^2 \tag{3.3}$$

$$\mathrm{Adv}_{\Phi_{kc,2d+2}}^{\mathsf{sprp}}(q) \leq \frac{kd}{2^{c-1}} q^2 \tag{3.4}$$

Thus, we have a classical security proof for the choice of the underlying `EGFN` used in Lilliput-TBC. Note that, in our case, c is equal to 8.

3.2.2 The π Permutation

Full diffusion delay is closely related to some structural attacks such as impossible differentials or integral attacks, as shown in [57, 8]. As there are many EGFNs that achieve $d = 4$, we chose one by taking other considerations into account. Specifically, we chose the block-wise permutation to maximize resistance against differential and linear cryptanalysis.

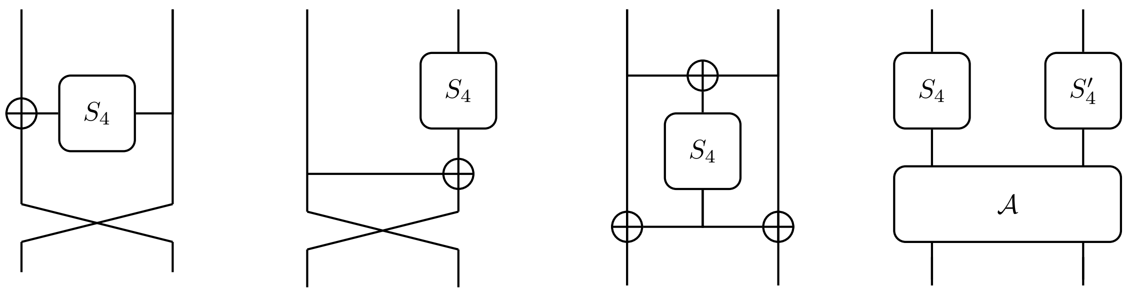
We identified the criterion for a permutation π to achieve $d = 4$ to be as follows: first, π must swap the 8 right-most blocks with the 8 left-most, and second, π must specifically swap blocks Y_7 and Y_{15} (a complete proof could be found in [8, 6]).

Up to block reindexing equivalence, there are exactly 37108 such permutations. For each of them, we computed the minimal number of differentially and linearly active S-boxes up to 20 rounds (see Section 3.4.1 and Table 3.2 for more details) and picked the one that maximizes the number of active S-boxes on 18, 19 and 20 rounds.

3.2.3 The S-box

Overall structure

We chose to build the 8-bit S-box from smaller ones so that it could be implemented with a fewer number of gates, which is a valuable property for hardware and bit-sliced software implementations. S-boxes built in this fashion usually rely on one of the four constructions depicted in Fig 3.1. We defined the following



(a) Feistel (b) MISTY (c) Lai-Massey (d) SPN

Figure 3.1: Some structures to build $2n$-bit S-boxes from n-bit ones.

selection criteria for the candidate S-box:

- Differential uniformity $\delta \leq 10$
- Linearity $\mathcal{L} \leq 64$
- Algebraic degree $\mathsf{deg} \geq 6$.

All constructions discussed above need to be iterated several times in order to achieve the desired cryptographic properties. Regarding Feistel and MISTY networks, [17] gives lower bounds on the differential uniformity and linearity for 3-round balanced constructions. In this case, Feistel networks provide better cryptographic properties as it is possible to reach $\delta = 8$ and $\mathcal{L} = 64$ versus $\delta = 16$ and $\mathcal{L} = 64$ for MISTY networks. It comes from the fact that Feistel networks do not require inner 4-bit S-boxes to be permutations, allowing the use of Almost Perfect Nonlinear (APN) functions as inner components. Still, it has been shown that 3-round unbalanced MISTY networks (e.g., dividing the 8-bit input into two inequal parts of 3 and 5 bits) can be used to build 8-bit S-boxes with $\delta = 8$ [17]. However, because the unbalanced words induce components with strong unbalanced degrees for the ANF, we decided to rule out this option.

Regarding the Lai-Massey scheme, the family of block ciphers FOX [34] uses a 3-round iterated structure in order to build an 8-bit S-box with $\delta = 16$ and $\mathcal{L} = 64$. On the other side, some cryptographic primitives simply add a nonlinear layer (i.e. two 4-bit S-boxes in parallel) at the beginning and/or at the end of the original scheme depicted in Fig. 3.1c instead of using an iterated structure and therefore save some XOR gates. For instance, the block cipher FLY [36] adds a nonlinear layer at the end of the scheme while the hash function Whirlpool [3] also adds one at the beginning resulting in a total of three and five inner 4-bit S-boxes, respectively. As for the MISTY ladder, the Lai-Massey structure requires inner 4-bit S-boxes to be permutations and therefore constructions that use three 4-bit S-boxes cannot reach a differential uniformity as good as Feistel networks. Although the cryptographic properties achieved by the variant using five 4-bit S-boxes are compliant with our selection criteria (i.e. $\delta = 8$, $\mathcal{L} = 56$ and $\mathsf{deg} = 7$ for the Whirlpool S-box), it is not worth the implementation cost.

The same reasoning can be applied to SPNs. For instance, the block cipher CLEFIA [55] uses two different 8-bit S-boxes and one of them relies on an SPN structure as defined in Fig. 3.1d with an additional nonlinear layer after $\mathcal{A}$ which refers to a matrix multiplication over $\mathbb{F}_{16}$. It results in an S-box with $\delta = 10$, $\mathcal{L} = 56$ and $\mathsf{deg} = 6$ which is compliant with our selection criteria. However, because it uses four 4-bit S-boxes, it is more heavy than a 3-round Feistel network to implement.

For all these reasons, we opted for an 8-bit S-box based on a 3-round balanced Feistel network. In the rest of this section, S_4^i refers to the inner 4-bit S-box at the i-th round.

Inner 4-bit S-boxes

According to [17], in order to reach $\delta = 8$, S_4^1 and S_4^3 have to be APN functions while S_4^2 has to be a permutation with differential uniformity 4. The authenticated block cipher SCREAM [26] uses an 8-bit S-box built in this manner where the underlying APN functions are $S_4^1 =$ `020b300a1e06a452` and $S_4^3 =$ `20b003a0e1604a25` and the permutation is $S_4^2 =$ `02c75fd64e8931ba`.

S_4^1 can be implemented using 11 instructions in total (either `AND` or `OR` or `XOR` or `NOT` or `MOV`), including 4 non-linear ones, while S_4^3 is directly derived from it by adding a `NOT` instruction in order to avoid fixed points. Although it is possible to find APN functions over $\mathbb{F}_{16}$ that are built using 10 instructions from the same instruction set, they all require at least 6 non-linear ones [17] which is not optimal regarding masked implementations. S_4^2 is built using 9 instructions from the same instruction set, including 4 non-linear ones, which is the smallest implementation cost for a 4-bit S-box with differential uniformity 4 [60]. Therefore, the SCREAM S-box allows very efficient implementations with and without masking as it only requires 44 instructions in total including 12 non-linear ones.

However, the number of non-linear operations is not the only criteria regarding Threshold Implementations (TI) where an S-box with algebraic degree d requires at least $n = d + 1$ shares. In order to limit the number of shares for a 4-bit S-box with $d > 2$, it has been proposed to use its decomposition into quadratic bijections [43] (i.e. $S_4^i = F \circ G$) so that it is possible to achieve a TI with $n = 3$. In order to fulfill the uniformity criteria, it has been proposed to find affine functions A_1 and A_2 such that $F = A_1 \circ \mathcal{Q} \circ A_2$, so that when it is possible to achieve a uniform sharing of the quadratic function Q, applying A_1 and A_2 on all input and output shares respectively gives a uniform sharing of F [11].

In [15] the authors study first-order TIs for several 8-bit S-boxes, including the one used in SCREAM. It results that the two APN functions S_4^1 and S_4^3 can be directly decomposed into two quadratic functions while the permutation S_4^2 requires affine functions as described above. In order to achieve more efficient TIs by saving the implemention cost of the affine functions, we looked for (and found) alternatives to S_4^2 that could be directly decomposed into two quadratic functions.

We chose to investigate all possible circuits with a Breadth-First Search (BFS) approach, including only `AND`, `XOR` and `NOT` gates as they can be straightforwardly thresholded. This approach is very similar to [60]. We optimized the number of gates used without considering `MOV` instructions as we consider that wiring is free compared to the cost of the gates. We allowed 5 registers during the exploration. Keeping the affine equivalence notion of the previous paper, stopping the exploration to 8 gates allowed us to reach 62 affine equivalence classes, including 3 optimal classes according to [50]. Following the same notation as [11] to refer to the equivalence classes, the three optimal classes we reached are $\mathcal{C}_{223}, \mathcal{C}_{296}$ and $\mathcal{C}_{297}$.

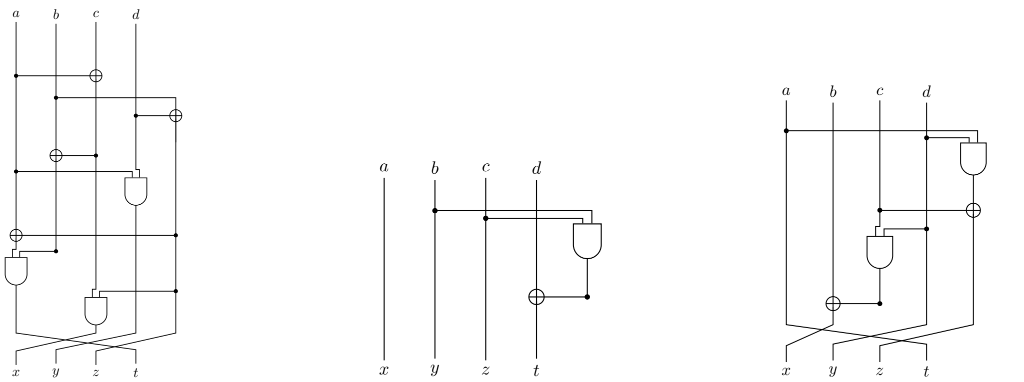
We focused on permutations of the optimal classes as they are the only ones with differential uniformity equal to 4. First, we eliminated candidates that did not allow to reach the selection criteria for the 8-bit S-box when used as S_4^2 in a 3-round Feistel network. After this step, there were still candidates in each optimal class. In order to go further into the optimization of TIs, we investigated the decomposition of the remaining candidates. Following [11], we decomposed those cubic permutations into two quadratic functions. There are six quadratic classes denoted by $\mathcal{Q}_4$, $\mathcal{Q}_{12}$, $\mathcal{Q}_{293}$, $\mathcal{Q}_{294}$, $\mathcal{Q}_{299}$ and $\mathcal{Q}_{300}$. It results from our BFS exploration that these classes can be implemented with a minimum of 2, 4, 6, 4, 6 and 6 gates, respectively. Among these classes, only $\mathcal{Q}_4$, $\mathcal{Q}_{294}$ and $\mathcal{Q}_{299}$ contain permutations that are uniform using direct sharing. However, neither $\mathcal{C}_{223}$ nor $\mathcal{C}_{296}$ nor $\mathcal{C}_{297}$ can be decomposed using $\mathcal{Q}_4$. On the other hand, because only $\mathcal{C}_{223}$ can be decomposed into two quadratic functions of the class $\mathcal{Q}_{294}$ that are uniform using direct sharing, this makes $\mathcal{C}_{223}$ the most interesting optimal class we reached regarding TIs.

As stated before, contrary to S_4^2, our aim was to avoid linear permutations between the quadratic functions. As we consider that wire permutations ω_i are free, for all the remaining candidates C in $\mathcal{C}_{223}$, we looked for 4-gate circuits Q_i and Q_j of $\mathcal{Q}_{294}$ that are uniform using direct sharing, such that $C = \omega_1 \circ Q_i \circ \omega_2 \circ Q_j \circ \omega_3$. As a final step to determine which composition to use, we calculated the cost (considered in Gate Equivalents – GEs) of a 3-share TI of all of them using this formula:

$$GE = 3GE_X \cdot X + (6GE_X + 9GE_A) \cdot A + GE_N \cdot N \tag{3.5}$$

with GE_X the area and X the number of `XOR` gates, GE_A the area and A the number of `AND` gates and GE_N the area and N the number of `NOT` gates. It comes from the fact that, when considering 3-share TIs, thresholding an `XOR` gate requires 3 `XOR` gates while thresholding an `AND` gate requires 6 `XOR` and 9 `AND` gates. Taking $GE_X = \frac{8}{3}$, $GE_A = \frac{4}{3}$ and $GE_N = \frac{2}{3}$, we found the minimum at $72 \cdot 2 = 144GE$s. Note that it does not include the cost of registers between the two permutations that are needed to ensure security against glitches.

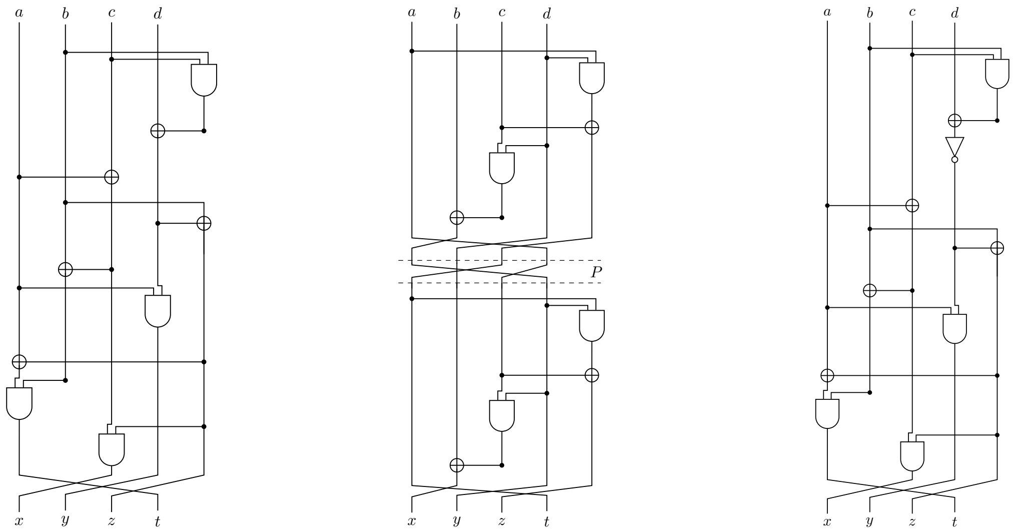
Among the several compositions that can be implemented using $144GE$s, we chose the permutation illustrated in Fig. 3.3b as it constitutes the only candidate that results from the composition of the same quadratic permutation $Q =$ `042e8ca6173d9fb5`, allowing to optimize the area cost in particular cases.



(a) $F =$ `020b30a01e06a425` (b) $G =$ `0123457689abcdfe` (c) $Q =$ `042e8ca6173d9fb5`

Figure 3.2: Quadatric functions used to build the cubic 4-bit S-boxes.

To put it in a nutshell, our 8-bit S-box is obtained by combining the two APN functions from the SCREAM S-box with the 4-bit permutation $\bar{S}_4^2 =$ `081f4c792b36e5da` in a 3-round Feistel network and achieves $\delta = 8$, $\mathcal{L} = 64$ and $\texttt{deg} = 6$ without fixed points. Thanks to the BFS exploration, we ensure that our S-box requires a small number of gates and that its TI is efficient as it uses the smallest circuits of its possible decompositions and futhermore, it does not require the use of affine permutations when decomposed into two quadratic functions. The 4-bit S-boxes are depicted in Fig. 3.3 while the underlying quadratic functions are depicted in Fig. 3.2, where a and d refer to the most and the least significant bits, respectively.



(a) $S_4^1 = F \circ G$ `020b300a1e06a452` (b) $\bar{S}_4^2 = Q \circ P \circ Q$ `081f4c792b36e5da` (c) $S_4^3 = F \circ (\oplus 1) \circ G$ `20b003a0e1604a25`

Figure 3.3: The three inner 4-bit S-boxes.

3.2.4 The Tweakey Schedule

As done for some other Tweakable Block Ciphers, we first looked at the `TWEAKEY` construction of [32] that fills p lanes of n bits divided into m words of c bits with the concatenation of the tweak T and of the key K. Then, to produce the subtweakey of each round, the `TWEAKEY` framework applies, on each lane, a permutation h acting on the m words and then multiply each of the m elements of c bits by a primitive root $\alpha_i, \forall i \in \{0, \cdots, p-1\}$ over $GF(2^c)$ different for each lane. Then, the subtweakey is the XOR of the p lanes and of a constant. From this construction that could be seen as the tensorial product of m Vandermonde matrices, the authors could deduce that the number of cancellations on $r+1$ subtweakeys is at most equal to $(p-1)$. Indeed, the updating function (excluding the h permutation) for the c bits words could be written as the following Vandermonde matrix

$$V = \begin{pmatrix} \alpha_0^0 & \alpha_0^1 & \alpha_0^2 & \cdots & \alpha_0^r \\ \alpha_1^0 & \alpha_1^1 & \alpha_1^2 & \cdots & \alpha_1^r \\ \alpha_2^0 & \alpha_2^1 & \alpha_2^2 & \cdots & \alpha_2^r \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{p-1}^0 & \alpha_{p-1}^1 & \alpha_{p-1}^2 & \cdots & \alpha_{p-1}^r \end{pmatrix}$$

when all α_i with $0 \leq i \leq r$ are distinct considering that $r < ord(\alpha)$. In this case, the code defined by V is a Reed-Solomon code of length $r+1$ and dimension p over $GF(2^c)$ and it is known to be MDS (Maximum Distance Separable). It means that its minimum distance is equal to $r+1-(p+1)$.

For designing our own tweakey schedule, we adopted the same idea to keep the Vandermonde strategy in order to guarantee the maximal possible number of cancellations. However, as we wanted to reduce the latency of the tweakey schedule and thus the number of computations, we adopted the following strategy instead of considering a lane as a vector of m elements of $GF(2^c)$: We directly consider the field $GF(2^{cm})$ that will be equal in our case to $GF(2^{64})$. Indeed, in our case, the size of each lane is equal to $\frac{n}{2} = 64$ bits due to the use of a Feistel-like scheme requiring only $\frac{n}{2} = 64$ bits of tweakey injected in the round function at each iteration.

Thus, we consider the p 64-bit long lanes as p elements of $GF(2^{64})$ and we multiply each lane by p different α_i given in Section 2.3.3 in a byte oriented matrix representation. Thus, with our construction, we obtain the following Vandermonde matrix constructed on $GF(2^{64})$:

$$V' = \begin{pmatrix} \alpha_0^0 & \alpha_0^1 & \alpha_0^2 & \cdots & \alpha_0^{r-1} \\ \alpha_1^0 & \alpha_1^1 & \alpha_1^2 & \cdots & \alpha_1^{r-1} \\ \alpha_2^0 & \alpha_2^1 & \alpha_2^2 & \cdots & \alpha_2^{r-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{p-1}^0 & \alpha_{p-1}^1 & \alpha_{p-1}^2 & \cdots & \alpha_{p-1}^{r-1} \end{pmatrix}$$

Thus, as $\forall 0 \leq i < r$, we have chosen the α_i such that $r < ord(\alpha_i)$, we preserve the MDS property induced by the underlying Reed-Solomon code and guarantee that the minimum distance is equal to $r - (p+1)$ leading to at most $(p-1)$ cancellations on r subtweakeys, seen always, as the XOR of the p lanes. This last choice is a logical one as done in many lightweight block ciphers, such as PRESENT [14], TWINE [58], LBlock [62] or SIMON [4] where the tweakey/key material is loaded in an initial register that is sequentially updated and where the subtweakeys/subkeys are extracted from that register.

We also chose to split the tweakey into p lanes of 64-bit instead of having a big state of $p \times 64$ bits and to update in parallel those p registers because small updating functions mix their content faster and increase performance. The downside is that each updating function could be attacked independently if their contents were not combined back during the subtweakey extraction which is not the case here.

Let us now explain how we have chosen the different α_i and the word-ring-LFSRs matrix multiplications at binary level that perform those operations.

We used LFSRs inspired by the results of [7] and [1] on LFSRs. LFSRs are typically used either in Fibonacci or Galois mode. In the first case, many feedbacks are used to influence a single cell while in the second case a single feedback influences many cells. In [7], the authors generalize LFSR beyond Fibonacci/Galois representation by allowing any cell to be used as feedback in any other cell. They call these new LFSRs "ring-LFSRs" because of the rotation occuring at each update. As the LFSRs in [1], the LFSRs chosen here have also a word-oriented structure: instead of performing bit-wise shift at each iteration and having binary feedbacks, they are shifted by one word at each update. As for the feedbacks, they are also word-oriented: one whole word is xored to another after possibly being transformed by a software-friendly operation such as shift or rotation. Those LFSRs are called word-LFSRs by their authors [1]. When a LFSR is both a word and ring LFSR, we call it a word-ring-LFSR. At the same time, they act at word level and they have feedbacks going from some word to some other. Word-ring-LFSRs have thus a smaller diffusion delay than classical Fibonacci or Galois LFSRs.

We have chosen our word-ring-LFSR defined by the matrix M for the α_1-multiplication with the minimal possible number of shift operations (3 at 8 bits words level) to minimize the number of XOR gates, with a primitive polynomial of degree 64. We chose words of size 8 bits to fit well on software platforms. Then, α_2 and α_3 multiplications are deduced directly using M^2 and M^3.

Moreover, to construct the matrix M_R for the α_4 multiplication, we searched for a matrix with 3 shift operations implementing the reciprocal (primitive) polynomial that defines M to ensure that the matrix V' stays a Vandermonde matrix and that the sequences produced when multiplying by α_1 (α_2 and α_3 respectively) and α_4 (α_5 and α_6 respectively) have only a single common value.

We have also chosen the different α_i with a primitive retroaction polynomial to ensure that the induced periods are maximal: the period for $\alpha_1, \alpha_2, \alpha_4$ and α_5 is maximal and equal to $2^{64} - 1$ whereas the period for α_3 and α_6 is equal to $\frac{2^{64}-1}{3}$.

Moreover, with this design strategy in mind, we are sure that the entire possible space is reached discarding the risk of an invariant attack as detailed in Section 3.4.

3.3 Security Analysis of the Modes of Operation

3.3.1 ΘCB3

The past year has seen several breakthroughs in the analysis of OCB, starting in October 2018 with the description by Inoue and Minematsu of a practical existential forgery attack [29]. Few weeks after, Poettering [45] extended this result and broke the confidentiality of OCB2, result that was extended further by Iwata [30] who devised a plaintext recovery attack[1]. These attacks were clearly announced by their authors as not applicable to OCB1 and OCB3, so ΘCB3 is also safe. To the best of our knowledge, no attacks were devised on ΘCB3.

3.3.2 SCT-2

To the best of our knowledge, no flaws were found so far in SCT-2 and the results published on Deoxys [63, 41, 18] only target the underlying cipher (that is, Deoxys-BC). In [18], the authors briefly discuss if their attacks on Deoxys-BC could apply once the cipher is used in the corresponding mode, and *"argue that [their] attacks are difficult to extend to Deoxys-II"*. This seems to indicate that the SCT-2 mode does not induce additional flaws to a cipher but on the contrary results in an extra protection coming from the fact that the attacker cannot access the decryption primitive.

To further support that the mode SCT-2 is trusted by the community, we recall here that Deoxys-II was selected after a 5-year process as the first choice for use case 3 ("Defense in depth") in the final Caesar portfolio [16].

3.3.3 Security Claims for the Modes

Our security claims for the different variants of Lilliput-AE are provided in Table 3.1.

	Security (bits)	
Goal (nonce-respecting case)	Lilliput-I	Lilliput-II
Key recovery	k	k
Confidentiality for the plaintext	n	$n-1$
Integrity for the plaintext	n	$n-1$
Integrity for the associated data	n	$n-1$

	Security (bits)	
Goal (nonce-misuse case)	Lilliput-I	Lilliput-II
Key recovery	k	k
Confidentiality for the plaintext	none	$n/2$
Integrity for the plaintext	none	$n/2$
Integrity for the associated data	none	$n/2$

Table 3.1: Security goals of Lilliput-AE in the nonce-respecting case and in the nonce-misuse case.

The bounds are given in the case of a tag size $\tau \geq n$. Should a smaller tag size be used, the security claims will drop according to τ. We derived the security bounds from the security proofs of ΘCB3 [39] and SCT [44] and we refer to them for more details.

3.4 Security Analysis of Lilliput-TBC

We analyze the security of Lilliput-TBC regarding classical attacks in the unknown key model and also in the related key model always considering the related tweak model. We will place ourselves for all

[1] These three results have been recently merged together in [28]

the attacks in the so-called "paranoid"' case, where the worst case is always envisaged even if it could not be reached.

Thus, we divide this section in the following way: we first consider differential/linear cryptanalysis, thus, extending those first results to the case of related key boomerang attacks and then, we give overall bounds for the so-called structural attacks that include impossible differential attacks, zero-correlation attacks, integral attacks and meet-in-the-middle attacks. Then, we take a particular attention on the following special attacks: division property, subspace cryptanalysis, algebraic attack.

Thus, we will first introduce the following bounds that will be used in the rest of this section:

- As the full diffusion is reached after $d = 4$ rounds for Lilliput-TBC, it means that no structural distinguisher can be constructed for more than $2d + 2$ rounds (see [8] for a detailed analysis and the security proofs).
- We will also always consider that the number of rounds that can be added to the best distinguisher for the key recovery part at the beginning is equal to d and at the end is also equal to d. Indeed, if a property is found on a single byte at the beginning or at the end of the distinguisher then after d rounds, all the input/output bytes will be influenced, so a key recovery could not exceed those bounds.
- As the tweakey schedule is fully linear and based on the XOR of elements of a Vandermonde matrix, it means that by reversing the linear system, one is able to find in the related tweak/related key models $(p - 1)$ cancellations (when p lanes are considered) placed at the best for the attacker.

First, let us precise that to prevent slide attacks [12] and as usually done in other tweakable block cipher proposals, different round constants are added to each subtweakey during the tweakey schedule process. So, we consider Lilliput-TBC immune to slide attacks.

3.4.1 Differential / Linear Cryptanalysis

To prove the resistance of Lilliput-TBC against differential and linear cryptanalysis, we give in Table 3.2 the lower bounds on the minimal number of active S-boxes in the single tweakey model considering no difference in the tweak. Those bounds partly fit with the ones given in [51] for the block cipher Lilliput. We have obtained those results using Constraint Programming up to 20 rounds in the single tweakey model. Due to the complexity of the tweakey schedule, we could not derive bounds for the related tweakey models (note that the related tweakey models are not considered for linear cryptanalysis). However, we could place ourselves in the worst case saying that authorizing a particular difference in a single lane i means that the results given in Table 3.2 on r rounds apply for $r + 2$ rounds, in two lanes i and j means that the results given in Table 3.2 on r rounds apply for $r + 3$ rounds, and so on up to p lanes are activated.

Moreover, we use here the fact that, as mentioned in Section 3.2.3, we have $\delta = 2^{-5}$ and $\mathcal{L} = 64$ for the chosen S-box.

Thus, with this reasoning, we could derive the following bounds for the different key lengths on the best differential/linear attacks:

- Lilliput-TBC-I-128 ($t = 192$, $k = 128$):
  - Best differential distinguisher on 13 rounds when no difference are introduced at all in the tweakey. Best possible differential attack on $13 + d + d = 13 + 8 = 21$ rounds in the same context. If a difference is introduced in b lanes, then the best attack is on $21 + b + 1$ rounds, leading to the best possible differential attack when the p lanes have differences equal to $21 + 5 + 1 = 27$ rounds.
  - With the same reasoning, the best linear distinguisher is on 16 rounds. Then, the best possible linear attack is on $16 + d + d = 16 + 8 = 24$ rounds.
- Lilliput-TBC-I-192 ($t = 192$, $k = 192$):
  - Best differential distinguisher on 17 rounds when no difference are introduced at all in the tweakey. Best possible differential attack on $13 + d + d = 17 + 8 = 25$ rounds in the same

context. If a difference is introduced in b lanes, then the best attack is on $25 + b + 1$ rounds, leading to the best possible differential attack when the p lanes have differences equal to $25 + 6 + 1 = 32$ rounds.

  - With the same reasoning, the best linear distinguisher is on 23 rounds (extrapolating the results of Table 3.2 up to 48 active S-boxes). Then, the best possible linear attack is on $23 + d + d = 23 + 8 = 31$ rounds.

- Lilliput-TBC-I-256 ($t = 192$, $k = 256$):

  - Best differential distinguisher on 24 rounds when no difference are introduced at all in the tweakey (always extrapolating the results of Table 3.2 up to 51 active S-boxes). Best possible differential attack on $24 + d + d = 24 + 8 = 32$ rounds in the same context. If, a difference is introduced in b lanes, then the best attack is on $32 + b + 1$ rounds, leading to the best possible differential attack when the p lanes have differences equal to $32 + 7 + 1 = 40$ rounds.

  - With the same reasoning, the best linear distinguisher is on 30 rounds (extrapolating the results of Table 3.2 up to 64 active S-boxes). Then, the best possible linear attack is on $30 + d + d = 30 + 8 = 38$ rounds.

- Lilliput-TBC-II-128 ($t = 128$, $k = 128$): The bound for the differential distinguisher is the same than the one given for Lilliput-TBC-I-192: the best differential attack works on 21 rounds for the single tweakey model and on 26 rounds when the p lanes have differences. The bound for the linear cryptanalysis is the same than the one given for Lilliput-TBC-I-192: 24 rounds.

- Lilliput-TBC-II-192 ($t = 128$, $k = 192$): The bound for the differential distinguisher is the same than the one given for Lilliput-TBC-I-192: the best differential attack works on 25 rounds for the single tweakey model and on 31 rounds when the p lanes have differences. The bound for the linear cryptanalysis is the same than the one given for Lilliput-TBC-I-192: 31 rounds.

- Lilliput-TBC-II-256 ($t = 128$, $k = 256$): The bound for the differential distinguisher is the same than the one given for Lilliput-TBC-I-256: the best differential attack works on 32 rounds for the single tweakey model and on 39 rounds when the p lanes have differences. The bound for the linear cryptanalysis is the same than the one given for Lilliput-TBC-I-256: 38 rounds.

Table 3.2: Minimal number of active S-boxes for every round. AS_D corresponds to the minimal number of S-boxes reached for a differential attack. AS_L corresponds to the minimal number of S-boxes reached for a linear attack.

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
AS_D	0	1	2	3	5	9	12	15	17	19	22	24	25	28	29	31	34	40	41	43
AS_L	0	1	2	3	5	8	12	13	15	17	19	22	25	27	30	32	34	38	40	42

3.4.2 Related Tweakey Boomerang Attacks

As the attacker could introduce differences both in the tweak and in the key and as our tweakey schedule is linear and could be completely computed according the introduced differences, we could imagine that in a related tweakey boomerang attack, the attacker could find a forward differential trail with $(p - 1)$ rounds containing no difference and could also find a backward differential trail with $(p - 1)$ rounds without difference. Thus, always considering that the key recovery part at the top of the related tweakey boomerang distinguisher has $d = 4$ rounds and also $d = 4$ rounds at the bottom, we could construct a related tweakey boomerang attack containing $2 \cdot (p - 1) + 8$ rounds at the beginning and at the end, and b rounds in its middle part.

Let us determine how many rounds is b for the different key lengths and considering that between the first differential trail on E_0 and the second differential trail on E_1 with $e = E_1 \circ E_0$, we have 2 rounds for free, one because the best coefficient of the `BCT` is equal to 1 and one because Lilliput-TBC is a Feistel-like scheme. Thus, as in a related tweakey boomerang attack, we associated the probability p

of the differential trail for E_0 and q for the differential trail for E_1. Thus, we expect that the overall probability for $b-2$ rounds is p^2q^2.

Thus, for a 256-bit key, we want $p^2q^2 \leq 2^{-256}$ considering that using several keys and several tweaks we could go beyond the full codebook limit. This leads to $p \leq 2^{-64}$ considering that $p = q$. Thus, referring to Table 3.2 with $\delta_S = 2^{-5}$, the differential trail for E_0 has at most $64/5 = 12.8$ active S-boxes leading to a differential propagating on at most 7 rounds. We apply the same reasoning for E_1. Thus, the maximal number of rounds a related tweakey boomerang attack could reach is equal to $7+7+8+2\cdot(p-1)+2 = 36$ for Lilliput-TBC-I-256 and to $7+7+8+2\cdot(p-1)+2 = 34$ for Lilliput-TBC-II-256.

Thus, for a 192-bit key, we want $p^2q^2 \leq 2^{-192}$. This leads to $p \leq 2^{-48}$ considering that $p = q$ and to $48/5 = 9.6$ active S-boxes for both E_0 and E_1 leading to a differential propagating on at most 6 rounds. Thus, the maximal number of rounds a related tweakey boomerang attack could reach is equal to $6+6+8+2\cdot(p-1)+2 = 32$ for Lilliput-TBC-I-192 and to $6+6+8+2\cdot(p-1)+2 = 30$ for Lilliput-TBC-II-192.

Thus, for a 128-bit key, we want $p^2q^2 \leq 2^{-128}$. This leads to $p \leq 2^{-32}$ considering that $p = q$ and to $32/5 = 6.4$ active S-boxes for both E_0 and E_1 leading to a differential propagating on at most 5 rounds. Thus, the maximal number of rounds a related tweakey boomerang attack could reach is equal to $5+5+8+2\cdot(p-1)+2 = 28$ for Lilliput-TBC-I-128 and to $5+5+8+2\cdot(p-1)+2 = 26$ for Lilliput-TBC-II-128.

3.4.3 Structural Attacks

In this Subsection, we consider all the so-called structural attacks which include impossible differential attacks [10], zero-correlation attacks [13], integral attacks [19] and meet-in-the-middle attacks [20]. The security analysis of those attacks mainly depend on the diffusion delay d of the scheme as shown in [8, 57]. Indeed, no distinguisher could be found for structural attacks beyond $2d+2$ rounds as full diffusion is reached. Those notions are also mainly linked with the computation of the super-pseudo random permutation advantage of the underlying scheme as shown in [27, 8].

Thus, in the single tweakey model where no difference at all is injected through the tweakey schedule, the best distinguisher could be constructed on $2d+2$ rounds. To complete the attack, we could add for the key recovery part d rounds at the top of the distinguisher and d rounds at the bottom leading to a structural attack with a maximum of $4d+2$ rounds. For all instances of Lilliput-TBC, this leads to the possibility of covering at most 18 rounds for all the structural attacks considered here. Note that this bound is overestimated compared to the one provided in [52] concerning the particular case of an impossible differential attack on the block cipher Lilliput.

Moreover, in the related tweakey model where we consider that an adversary can control at most the content of p lanes, the adversary could directly in this context attack $4d+2+p$ rounds at most. This leads to the following upper bounds on the possible number of attacked rounds for the 6 instances of Lilliput-TBC: 22 rounds for Lilliput-TBC-II-128, 23 rounds for Lilliput-TBC-II-192 and Lilliput-TBC-I-128, 24 rounds for Lilliput-TBC-II-256 and Lilliput-TBC-I-192, 25 rounds for Lilliput-TBC-I-256.

3.4.4 Division Property

The division property was proposed by Todo [59] as a generalization of the integral property to correctly evaluate higher-order integral property. The best division distinguisher described in [53] on the block cipher Lilliput is on 13 rounds leading to a key recovery attack on 17 rounds in the single tweakey model. Note that the `Linear` procedure presented in Algorithm 1 of [53] is the same for Lilliput-TBC, only the `NonLinear` part diverges in the way to compute the sets. The distinguisher presented in [53] studies an integral property on 63 input bits and on 1 output bit that completely maximize the possible number of implied bits. Thus, we conjecture that there is no distinguisher that exploits division properties on more than 26 rounds of Lilliput-TBC as in this last case, the number of possible input bits implied in a division property is doubled, i.e. equal to 127 with the same procedure describing the linear part. Thus, we are still confident that our proposals offer a strong security margin regarding this class of attacks.

3.4.5 Subspace Cryptanalysis

Invariant subspace cryptanalysis uses affine subspaces that are invariant throughout the cipher. Those attacks work particularly well in the context of simple tweakey/key schedules where the invariant properties stay valid through the key addition. Thus, to avoid this kind of attacks, invariant subspaces must be destroyed by the key/tweakey schedule. As the tweakey schedule of Lilliput-TBC is composed of ring-LFSRs ranging all the possible spaces, we conjecture that our non-trivial tweakey schedule provide a good protection against those attacks.

However, we give here the 9 linear structures of our S-box, i.e. the list (b, a, c) such that $b \cdot (S(x) \oplus S(x \oplus a)) = c$ with c a constant: (1, 32, 1), (1, 64, 0), (1, 96, 1), (4, 64, 1), (4, 128, 0), (4, 192, 1), (5, 64, 1), (5, 160, 1), (5, 224, 0). Note that none of those structures is preserved through two applications of the S-box.

Thus, with our non-trivial tweakey schedule and the fact that the invariant subspaces deduced from the linear structures can not be chained for many rounds, we conjecture that Lilliput-TBC is immune against this kind of attacks.

3.4.6 Algebraic Attacks

Before deducing bounds for algebraic attacks, let us describe the algebraic properties of the S-box. The S-box has a maximal degree of 6 and a minimal degree of 4. Our S-box could also be described using $e = 14$ quadratic equations in the 16 input/output variables over $GF(2)$. Thus, from Table 3.2, we could see that for 13 rounds, we have 26 active S-boxes, it means that, from this bound the number of induced variables by the algebraic expression of the cipher Lilliput-TBC is greater than the block size.

Moreover, as our S-box S could be described with 14 quadratic equations in 16 variables, it means that the number of quadratic equations induced by a round is $14 \times 8 = 112$ quadratic equations in $16 \times 8 = 128$ variables and for 32 rounds of Lilliput-TBC, we thus obtain 3584 quadratic equations in 4096 variables. Thus, we obtain an under-determined system with more variables than for the AES.

Using those arguments, we conjecture that Lilliput-TBC is immune against algebraic attacks.

3.4.7 Differential Fault Analysis in Middle Rounds

We want to protect Lilliput-TBC against differential fault analysis. Such attacks consist in injecting faults in one of the last rounds of the encryption, and exploit pairs of faulty and correct ciphertexts. A common countermeasure consists in doubling the execution of a few last rounds in order to detect a fault. In the case of a fault injection – unless the attacker is able to inject twice the same fault in a very short period of time – the doubling results in two ciphertexts, one faulty and the other not. Such a result is detected and no output is given. In order to be detected, the fault must be injected during the rounds that are doubled. If a fault occurs before, the faulty state is copied and processed twice, resulting in two identical faulty ciphertexts which will be outputted, making the countermeasure ineffective. For this reason, it is important to protect enough rounds to prevent such attacks. It is also important to evaluate closely the number of rounds to protect as doubling increases time computation or surface area.

We analyze how much rounds must be protected in order to prevent the attack from Rivain [47] adapted to Lilliput-TBC. This attack takes advantage of the Feistel scheme in order to inject fault in middle rounds and observe differential distributions in the last one.

As Lilliput-TBC is based on a Feistel scheme, we will denote the state at output of round i as (L^i, R^i), L^i and R^i being its 64-bit left and right parts respectively[2]. Hence, the plaintext is (L^0, R^0), and the ciphertext (L^r, R^r). The fact that the number of rounds r changes with the mode used does not change anything about the following results, as r is always greater than the number of rounds to protect. Notice that the subtweakey byte used in each non linear function F_j of Lilliput-TBC $(0 \leq j \leq 7)$ is the XOR of a known constant and p byte values $(4 \leq p \leq 7)$ that come from some known tweak-dependent lanes and other unknown key-dependent ones. The following analysis focuses on retrieving this subtweakey byte value only, leaving uncertainty about key-lane bytes. A "successful" attack thus leaves the attacker with 2^8 pairs of key-dependent bytes (or 2^{16} triplets or 2^{32} quadruplets, depending on the key length).

[2] With notations of Section 2.3.1 we have $L = (X_{15}, \ldots, X_8)$ and $R = (X_7, \ldots, X_0)$.

As in [47] we only consider faults in the left part of the state as it is the most efficient way to retrieve the subkey in a Feistel scheme. Injecting a fault ϵ in L^i induces changes in the next rounds and results in a faulty ciphertext $\widetilde{C}$. The correct ciphering of the same plaintext is denoted C. The main goal is to observe the distribution of $\Delta = L^{r-1} \oplus \widetilde{L}^{r-1}$, which is the XOR of both left parts of the correct and faulty states before the last round. It is possible to compute each byte of $\Delta = (\delta_7, \ldots, \delta_0)$ as a function of the correct/faulty ciphertexts and a guess g on the corresponding subtweakey byte in the last round. Denoting $L^i = (\ell_7^i, \ldots, \ell_0^i)$ and $\widetilde{L}^i = (\widetilde{\ell}_7^i, \ldots, \widetilde{\ell}_0^i)$ – and similarly with r_j^i and $\widetilde{r}_j^i$ for the right parts – we infer:

$$\delta_0 = \ell_0^r \oplus \widetilde{\ell}_0^r \oplus S(r_7^r \oplus g) \oplus S(\widetilde{r}_7^r \oplus g) \tag{3.6}$$

$$\delta_j = \ell_j^r \oplus \widetilde{\ell}_j^r \oplus S(r_{7-j}^r \oplus g) \oplus S(\widetilde{r}_{7-j}^r \oplus g) \oplus r_7^r \oplus \widetilde{r}_7^r \qquad \text{for } 1 \leq j \leq 6 \tag{3.7}$$

$$\begin{aligned} \delta_7 = {} & \ell_7^r \oplus \widetilde{\ell}_7^r \oplus S(r_0^r \oplus g) \oplus S(\widetilde{r}_0^r \oplus g) \oplus r_7^r \oplus \widetilde{r}_7^r \\ & \oplus r_6^r \oplus \widetilde{r}_6^r \oplus r_5^r \oplus \widetilde{r}_5^r \oplus r_4^r \oplus \widetilde{r}_4^r \oplus r_3^r \oplus \widetilde{r}_3^r \oplus r_2^r \oplus \widetilde{r}_2^r \oplus r_1^r \oplus \widetilde{r}_1^r \end{aligned} \tag{3.8}$$

Depending on the round where the fault is injected, the attacker might be able to know the distribution of Δ. For example, if a fault ϵ is injected in L^{r-3}, then $\Delta = \epsilon$. If the attacker knows ϵ, he can check whether the g-dependent Δ value calculated from the previous equations equals ϵ or not, discarding subtweakey candidates that do not. Note that due to the byte oriented scheme of Lilliput-TBC, the attack can be done on each subtweakey byte independently, allowing to guess one byte and calculate one δ byte at a time. For this reason, the rest of the analysis focuses on a byte δ rather than on the whole Δ.

If the attacker is able to systematically fault with the same known ϵ in earliest rounds, he can build (in an offline phase) approximations of the theoretic distribution of any δ_j. Given N pairs of correct/faulty ciphertexts $(C, \widetilde{C})_N$, the attack then consists in calculating, for each candidate g, the corresponding empirical distribution of δ_j and select the one that is the most similar to the theoretic one. This can be done in a maximum-likelihood manner for instance.

In the case where the attacker is not able to predict the distribution of δ_j, he can still expect it to be biased for the correct key guess, and to be uniform for others (wrong-key assumption). Similarly he can compute the empirical distributions of δ_j and select the one that is the farthest from the uniform distribution. This can be done with the Squared Euclidean Imbalance distinguisher for example.

In order to infer the number of rounds to protect, we have considered a strong attacker who is able to inject bit flips at the bit position of its choice. We then conducted attacks, injecting faults sooner and sooner until the attack becomes unfeasible.

Simulation Results and Recommendation We have simulated the differential fault analysis where a bit flip fault is injected at a precise round $r - s$ and at a chosen bit position b on Lilliput-TBC-I-128[3]. We have studied both the non profiled case where the distinguisher is the Squared Euclidean Imbalance of the observed distribution of δ_j, and the profiled case based on the maximum-likelihood of this distribution with respect to (an approximation of) the theoretic one. For sake of clarity, with our notations, when the attacker observes the distribution of byte δ_j, it helps him to recover the subtweakey byte RTK_{7-j}. Our objective is to determine the largest value of s for which we suspect that a fault attack can be realized. The fault bit position b is numbered from 0 to 63 which respectively denote the least significant bit of $X_8 = \ell_0$ and the most significant bit of $X_{15} = \ell_7$. For any set of parameters (round gap s, fault bit position b, attacked subtweakey byte position $(7 - j)$, number of faults N, profiled/non profiled setting), our results are expressed as the average success rate on 1000 runs.

We first observed that for $s \leq 6$, the fault attack is somewhat easy. For $s = 6$, and for $N = 1000$ faults, there always exists a fault bit position for which the success rate is 1.0 for all positions j except for $j = 7$. Note that we have systematically observed that the subtweakkey byte number 0 ($j = 7$) is the most difficult to retrieve. For $j = 7$, the success rate may still be as large as 0.873 (depending on b) in the profiled case. An interesting observation is that the success of the attack greatly depends on the fault bit position. As we consider that the attacker can choose b, we think that the relevant criteria is the maximum success rate taken on all $b = 0 \ldots 63$ values.

Tables 3.3 and 3.4 present results for the non-profiled and the profiled settings respectively. We have considered a number of faults N belonging to the set $\{10^3, 3.10^3, 10^4, 3.10^4, 10^5, 3.10^5, 10^6, 3.10^6, 10^7\}$ for

[3]We guess that similar results would have been obtained for other versions.

round	faults	attacked subtweakey byte : RTK_{7-j}							
		j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
$s=6$	10^3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.032
$s=7$	10^3	0.009	0.014	0.011	0.008	0.010	0.008	0.008	0.011
	3.10^3	0.008	0.078	0.010	0.009	0.011	0.009	0.021	0.008
	10^4	0.009	0.523	0.027	0.012	0.010	0.010	0.160	0.008
	3.10^4	0.022	0.764	0.303	0.030	0.013	0.011	0.684	0.011
	10^5	0.381	1.0	0.998	0.352	0.017	0.020	0.994	0.009
	3.10^5	1.0	1.0	1.0	0.506	0.064	0.130	1.0	0.009
	10^6	1.0	1.0	1.0	0.891	0.666	0.878	1.0	0.010
	3.10^6	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.008
	10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.008
$s=8$	10^5	0.010	0.008	0.011	0.011	0.008	0.009	0.012	0.009
	3.10^5	0.008	0.011	0.008	0.008	0.009	0.008	0.008	0.009
	10^6	0.009	0.009	0.010	0.013	0.010	0.012	0.008	0.009
	3.10^6	0.008	0.008	0.011	0.011	0.010	0.008	0.010	0.009
	10^7	0.012	0.009	0.009	0.008	0.009	0.008	0.008	0.011

Table 3.3: Experimental success rate of non profiled (Squared Euclidean Imbalance) differential fault analysis

round	faults	attacked subtweakey byte : RTK_{7-j}							
		j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
$s=6$	10^3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.837
$s=7$	10^3	0.036	0.448	0.076	0.023	0.012	0.013	0.226	0.009
	3.10^3	0.083	0.730	0.275	0.076	0.019	0.019	0.668	0.010
	10^4	0.336	0.990	0.825	0.291	0.046	0.053	0.961	0.009
	3.10^4	0.875	1.0	1.0	0.555	0.113	0.180	1.0	0.009
	10^5	1.0	1.0	1.0	0.728	0.497	0.669	1.0	0.012
	3.10^5	1.0	1.0	1.0	0.992	0.965	0.992	1.0	0.010
	10^6	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.011
	3.10^6	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.009
	10^7	-	-	-	-	-	-	-	-
$s=8$	10^5	0.010	0.007	0.009	0.007	0.009	0.011	0.010	0.011
	3.10^5	0.009	0.009	0.009	0.009	0.009	0.007	0.007	0.009
	10^6	0.009	0.012	0.008	0.011	0.010	0.010	0.009	0.011
	3.10^6	0.009	0.009	0.008	0.008	0.009	0.010	0.009	0.010
	10^7	0.009	0.008	0.011	0.008	0.010	0.009	0.008	0.012

Table 3.4: Experimental success rate of profiled (Maximum Likelihood) differential fault analysis

$s = 7$, and to the set $\{10^5, 3.10^5, 10^6, 3.10^6, 10^7\}$ for $s = 8$. Without surprise, one can observe that the profiled attack is more efficient than the non-profiled one. For $s = 7$ all subtweakey bytes except for $j = 7$ can be retrieved with about 10^5 faults. Even with only about 3000 faults two subtweakey bytes (for $j = 1$ and $j = 6$) can be recovered. We also observe that for $s = 8$ the attack does not work, even in the profiled case and even with ten millions faults, whatever the bit fault position and whatever the attacked byte.

Based on our simulation results, we recommend to protect Lilliput-TBC against differential fault analysis by doubling the execution of a minimum of seven last rounds.

3.4.8 Security Evaluation Summary

Table 3.5 gives a security evaluation summary for all the instances of Lilliput-TBC. From this table, we are able to say that each instance has a sufficient security margin (given in the last column).

	STKM			RTKM				Nb rounds	Sec. Margin
	Diff.	Lin.	Struct.	Diff.	Lin.	RTKB	Struct.	(r)	(in rounds)
Lilliput-TBC-I-128	21	24	18	27	24	28	23	32	4
Lilliput-TBC-I-192	25	31	18	32	31	32	24	36	4
Lilliput-TBC-I-256	32	38	18	40	38	36	25	42	2
Lilliput-TBC-II-128	21	24	18	26	24	26	22	32	6
Lilliput-TBC-II-192	25	31	18	31	31	30	23	36	5
Lilliput-TBC-II-256	32	38	18	39	38	34	24	42	3

Table 3.5: Security Evaluation summary ("paranoid" case). STKM means "Single Tweakey Model", RTKM means "Related Tweakey Model" and RTKB means "Related Tweakey Boomerang attack".

Surprisingly, classical attacks such as differential and linear attacks reach more rounds than structural attacks for Lilliput-TBC. This is mainly linked with the choice of a Feistel-like scheme with a good diffusion.

Chapter 4

Implementations

Lilliput-AE is suited to be implemented efficiently on a wide range of processors (especially those embedded in IoT platforms) and in hardware. This chapter will provide insights on efficient implementation methods, especially on 8-bit processors where Lilliput-AE is by design well adapted due to its byte-oriented nature. Many good properties on 8-bit platforms are also valid on 16-bit and 32-bit platforms.

4.1 Software Implementations

In this section, we give some possible variants for implementing Lilliput-AE. We take as reference IoT platforms those from the FELICS (Fair Evaluation of LIghtweight Cryptographic Systems) framework [21]: 8-bit Atmel AVR ATmega128, 16-bit Texas Instruments MSP430F1611 and 32-bit Arduino Due ARM Cortex-M3. When useful, binary code size, RAM, and execution time optimizations will be discussed.

On an 8-bit processor, Lilliput-AE can be programmed by simply implementing the different component transformations.

4.1.1 Round Function `OneRoundEGFN`

If we look at the `OneRoundEGFN` the round function, `NonLinearLayer` function is only made of S-boxes computation, with previous subtweakey addition. `LinearLayer` function is just successive byte additions on x_{15} followed by byte additions on most significant bytes of `OneRoundEGFN` internal state. Finally, a byte-oriented permutation, `PermutationLayer`, is computed.

A straightforward implementation is then easy to implement on 8-bit processors. The implementation of all the F_i functions requires a table of 256 bytes. Since this table is fixed, it can be easily stored in EEPROM data. As mentioned in section 3.2, S has been chosen to be easily masked in hardware and software.

Concerning code size, `OneRoundEGFN` can be easily computed with loops in order to save ROM program space. For example, algorithm 5 shows that `NonLinearLayer` only needs one additional intermediate register in total to store the successive results of $SK_{7-i} \oplus x_{7-i}$ and the S computation on it: RAM stack usage is then minimal.

Algorithm 5: x_{8+i} computation in `Non-Linear Layer`

for $i = 0$ **to** 7 **do**
  $x_{8+i} \leftarrow x_{8+i} \oplus S(SK_{7-i} \oplus x_{7-i})$
return $(x_{15}, x_{14}, \cdots, x_8)$

Similarly, algorithm 6 shows that `LinearLayer` can be also implemented by `XOR` additions and accumulations.

Algorithm 6: x_{8+i} computation in `Linear Layer`

```
for i = 0 to 7 do
    x_15 ← x_15 ⊕ x_i
for i = 0 to 6 do
    x_{14−i} ← x_{14−i} ⊕ x_7
return (x_15, x_14, ··· , x_8)
```

Name	k	t	p	Required $\boldsymbol{\alpha_i}$
Lilliput-TBC-I-128	128	192	5	α_0 to α_4
Lilliput-TBC-I-192	192	192	6	α_0 to α_5
Lilliput-TBC-I-256	256	192	7	α_0 to α_6
Lilliput-TBC-II-128	128	128	4	α_0 to α_3
Lilliput-TBC-II-192	192	128	5	α_0 to α_4
Lilliput-TBC-II-256	256	128	6	α_0 to α_5

Table 4.1: Multiplications needed for each variant of Lilliput-TBC

Finally, `PermutationLayer` can be simply implemented as a series of `MOV` operations. For example, in encryption mode: $x_{13} \leftarrow x_0$, $x_9 \leftarrow x_1$, $\cdots$, $x_7 \leftarrow x_{15}$.

Overall, a straightforward computation of `OneRoundEGFN` (which is the same for every TK size) needs 29 `XOR`s (21 in the datapath plus 8 before each S-box computation), 8 accesses in EEPROM memory for S-box computations, and 16 `MOV` operations for `PermutationLayer`, i.e. only 53 operations in total (29 arithmetic operations and 24 memory operations). The footprint on RAM stack, ROM program (as discussed earlier in this subsection) and on EEPROM data (256 bytes) of `OneRoundEGFN` is very lightweight for 8-bit platforms.

4.1.2 Tweakey Schedule

The tweakey schedule can be decomposed into two distinct functions:

- the *extraction* function, which is called r times to produce subtweakey RTK^i from the tweakey state TK^i, $\forall i \in \{0, \cdots, r-1\}$,
- the *update* function, which is called $r-1$ times to compute TK^{i+1} from TK^i, $\forall i \in \{0, \cdots, r-2\}$.

The update function consists in one multiplication α_i per lane. Each of these multiplications takes a different amount of operations to complete. Table 4.1 summarizes which multiplications are needed for each variant of Lilliput-TBC.

The extraction function consists in:

- xoring all p 64-bit lanes together bytewise: this requires $(p-1)$ 64-bit `XOR`s, hence $8 \times (p-1)$ 8-bit `XOR`s,
- xoring the resulting 64-bit word with the round constant C^i: this requires a single 8-bit `XOR`, since C^i fits on 8 bits.

This function thus requires $8 \times (p-1) + 1$ `XOR` operations.

4-lane case

The following multiplications are needed to process four lanes: $\alpha_0 = I$, $\alpha_1 = M$, $\alpha_2 = M^2$, and $\alpha_3 = M^3$. As we will do for further number of lanes, we will develop the matrix relations to evaluate precisely the number of required operations.

$\boldsymbol{\alpha_0}$ is the identity function, so it is a completely **free operation**.

Multiplication α_1 of vector $x = (x_7, x_6, \cdots, x_0)^t$ by matrix M can be expressed as:

$$\begin{pmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{pmatrix} = \begin{pmatrix} x_6 \\ x_5 \\ x_5 \ll 3 \oplus x_4 \\ x_4 \gg 3 \oplus x_3 \\ x_2 \\ x_6 \ll 2 \oplus x_1 \\ x_0 \\ x_7 \end{pmatrix}$$

Multiplication α_1 will thus require 14 operations in total:

- 3 shift operations,
- 3XORs,
- 8 assignments.

Multiplication α_2 is represented by matrix M^2, which corresponds to two successive applications of matrix M. Let us denote $M \cdot x$ as $x_M = (x_{M,7}, \cdots, x_{M,0})^t$:

$$\begin{pmatrix} x_{M,7} \\ x_{M,6} \\ x_{M,5} \\ x_{M,4} \\ x_{M,3} \\ x_{M,2} \\ x_{M,1} \\ x_{M,0} \end{pmatrix} = \begin{pmatrix} x_6 \\ x_5 \\ x_5 \ll 3 \oplus x_4 \\ x_4 \gg 3 \oplus x_3 \\ x_2 \\ x_6 \ll 2 \oplus x_1 \\ x_0 \\ x_7 \end{pmatrix}$$

$y = M^2 \cdot x$ can then be expressed as:

$$\begin{pmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{pmatrix} = \begin{pmatrix} x_{M,6} \\ x_{M,5} \\ x_{M,5} \ll 3 \oplus x_{M,4} \\ x_{M,4} \gg 3 \oplus x_{M,3} \\ x_{M,2} \\ x_{M,6} \ll 2 \oplus x_{M,1} \\ x_{M,0} \\ x_{M,7} \end{pmatrix}$$

Some components of x_M are simply permuted components of x; others (namely, $x_{M,5}$, $x_{M,4}$ and $x_{M,2}$) result from a linear combination of components of x. Some of these combinations contribute to more than one components of y: specifically, $x_{M_5} = x_5 \ll 3 \oplus x_4$ and $x_{M,4} = x_4 \gg 3 \oplus x_3$.

To minimize the number of operations, we can thus spend some registers to store $x_{M,5}$ and $x_{M,4}$. The final expression for $y = M^2 \cdot x$ then becomes:

$$\begin{pmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{pmatrix} = \begin{pmatrix} x_{M,6} \\ x_{M,5} \\ x_{M,5} \ll 3 \oplus x_{M,4} \\ x_{M,4} \gg 3 \oplus x_{M,3} \\ x_{M,2} \\ x_{M,6} \ll 2 \oplus x_{M,1} \\ x_{M,0} \\ x_{M,7} \end{pmatrix} = \begin{pmatrix} x_5 \\ x_{M,5} \\ x_{M,5} \ll 3 \oplus x_{M,4} \\ x_{M,4} \gg 3 \oplus x_2 \\ x_6 \ll 2 \oplus x_1 \\ x_5 \ll 2 \oplus x_0 \\ x_7 \\ x_6 \end{pmatrix}$$

Multiplication α_2 will thus require 22 operations in total:

- 2 XORs, 2 shifts and 2 assignments for $x_{M,5}$ and $x_{M,4}$,
- 4 XORs and 4 shifts,

- 8 byte assignments.

Multiplication α_3 is represented by matrix M^3, which corresponds to three successive applications of matrix M. Let us denote $M^2 \cdot x$ as $x_{M^2} = (x_{M^2,7}, \cdots, x_{M^2,0})^t$:

$$\begin{pmatrix} x_{M^2,7} \\ x_{M^2,6} \\ x_{M^2,5} \\ x_{M^2,4} \\ x_{M^2,3} \\ x_{M^2,2} \\ x_{M^2,1} \\ x_{M^2,0} \end{pmatrix} = \begin{pmatrix} x_{M,6} \\ x_{M,5} \\ x_{M,5} \ll 3 \oplus x_{M,4} \\ x_{M,4} \gg 3 \oplus x_{M,3} \\ x_{M,2} \\ x_{M,6} \ll 2 \oplus x_{M,1} \\ x_{M,0} \\ x_{M,7} \end{pmatrix}$$

$y = M^3 \cdot x$ can then be expressed as:

$$\begin{pmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{pmatrix} = \begin{pmatrix} x_{M^2,6} \\ x_{M^2,5} \\ x_{M^2,5} \ll 3 \oplus x_{M^2,4} \\ x_{M^2,4} \gg 3 \oplus x_{M^2,3} \\ x_{M^2,2} \\ x_{M^2,6} \ll 2 \oplus x_{M^2,1} \\ x_{M^2,0} \\ x_{M^2,7} \end{pmatrix}$$

As with α_2, we can isolate components of x_{M^2} which satisfy the following constraints:

1. they result from a linear combination of more than one components of x_M,
2. they contribute to more than one components of y.

$x_{M^2,5}$, $x_{M^2,4}$ and $x_{M^2,2}$ satisfy constraint 1; among those, only $x_{M^2,5} = x_{M,5} \ll 3 \oplus x_{M,4}$ and $x_{M^2,4} = x_{M,4} \gg 3 \oplus x_{M,3} = x_{M,4} \gg 3 \oplus x_2$ satisfy constraint 2. To implement α_3 using as few operations as necessary, we thus need to:

1. pre-compute $x_{M,5}$ and $x_{M,4}$,
2. pre-compute $x_{M^2,5}$ and $x_{M^2,4}$,
3. compute y as follows:

$$\begin{pmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{pmatrix} = \begin{pmatrix} x_{M^2,6} \\ x_{M^2,5} \\ x_{M^2,5} \ll 3 \oplus x_{M^2,4} \\ x_{M^2,4} \gg 3 \oplus x_{M^2,3} \\ x_{M^2,2} \\ x_{M^2,6} \ll 2 \oplus x_{M^2,1} \\ x_{M^2,0} \\ x_{M^2,7} \end{pmatrix} = \begin{pmatrix} x_{M,5} \\ x_{M^2,5} \\ x_{M^2,5} \ll 3 \oplus x_{M^2,4} \\ x_{M^2,4} \gg 3 \oplus x_{M,2} \\ x_{M,6} \ll 2 \oplus x_{M,1} \\ x_{M,5} \ll 2 \oplus x_{M,0} \\ x_{M,7} \\ x_{M,6} \end{pmatrix} = \begin{pmatrix} x_{M,5} \\ x_{M^2,5} \\ x_{M^2,5} \ll 3 \oplus x_{M^2,4} \\ x_{M^2,4} \gg 3 \oplus x_6 \ll 2 \oplus x_1 \\ x_5 \ll 2 \oplus x_0 \\ x_{M,5} \ll 2 \oplus x_7 \\ x_6 \\ x_5 \end{pmatrix}$$

Multiplication α_3 will thus require 30 operations in total:

- 2 `XOR`s, 2 shifts and 2 assignments for $x_{M,5}$ and $x_{M,4}$,
- 2 `XOR`s, 2 shifts and 2 assignments for $x_{M^2,5}$ and $x_{M^2,4}$,
- 5 `XOR`s, and 5 shifts,
- 8 byte assignments.

To sum up, **for the 4-lane case**, the multiplications by α_0, α_1, α_2 and α_3 require $0+14+22+30 =$ **66 operations in total**.

Taking into account the $8 \times (p-1)+1 = 8 \times 3+1 = 25$ operations needed for the extraction function, this leads to $66 + 25 =$ **91 operations for the whole subtweakey computation**.

5-lane case

To process five lanes, multiplications $\alpha_0 = I$, $\alpha_1 = M$, $\alpha_2 = M^2$, $\alpha_3 = M^3$ (already described) and $\alpha_4 = M_R$ are needed.

Multiplication α_4 of vector $x = (x_0, x_1, \cdots, x_7)^t$ by matrix M_R (as explained in section 2.3.3, we invert the direction of binary notations when dealing with M_R) can be also expressed as:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \oplus x_4 \ll 3 \\ x_4 \\ x_5 \oplus x_6 \gg 3 \\ x_3 \gg 2 \oplus x_6 \\ x_7 \\ x_0 \end{pmatrix}$$

Multiplication α_4 will thus require 14 operations in total:

- 3 `XOR`s and 3 shifts,
- 8 byte assignments.

To sum up, **for the 5-lane case**, the update function will require 66 operations (cf. 4-lane case) plus 14 operations for α_4, i.e. 80 operations. After adding $8 \times (5-1) + 1 = 33$ `XOR`s for the extraction function, we reach **113 operations for the whole subtweakey computation**.

6-lane case

To process six lanes, multiplications $\alpha_0 = I$, $\alpha_1 = M$, $\alpha_2 = M^2$, $\alpha_3 = M^3$, $\alpha_4 = M_R$ (already described) and $\alpha_5 = M_R^2$ are needed.

Multiplication α_5 of vector $x = (x_0, x_1, \cdots, x_7)^t$ by matrix M_R^2 corresponds to two successive applications of matrix M_R. Let us denote $M_R \cdot x$ as $x_{M_R} = (x_{M_R,0}, \cdots, x_{M_R,7})^t$:

$$\begin{pmatrix} x_{M_R,0} \\ x_{M_R,1} \\ x_{M_R,2} \\ x_{M_R,3} \\ x_{M_R,4} \\ x_{M_R,5} \\ x_{M_R,6} \\ x_{M_R,7} \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \oplus x_4 \ll 3 \\ x_4 \\ x_5 \oplus x_6 \gg 3 \\ x_3 \gg 2 \oplus x_6 \\ x_7 \\ x_0 \end{pmatrix}$$

$y = M_R^2 \cdot x$ can then be expressed as:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} x_{M_R,1} \\ x_{M_R,2} \\ x_{M_R,3} \oplus x_{M_R,4} \ll 3 \\ x_{M_R,4} \\ x_{M_R,5} \oplus x_{M_R,6} \gg 3 \\ x_{M_R,3} \gg 2 \oplus x_{M_R,6} \\ x_{M_R,7} \\ x_{M_R,0} \end{pmatrix}$$

As with α_2 and α_3, we will pre-compute components of x_{M_R} which depend on more than one components of x and contribute to more than one components of y. For α_5, this singles out $x_{M_R,4} = x_5 \oplus x_6 \gg 3$. We end up with the following expression for y:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} x_{M_R,1} \\ x_{M_R,2} \\ x_{M_R,3} \oplus x_{M_R,4} \ll 3 \\ x_{M_R,4} \\ x_{M_R,5} \oplus x_{M_R,6} \gg 3 \\ x_{M_R,3} \gg 2 \oplus x_{M_R,6} \\ x_{M_R,7} \\ x_{M_R,0} \end{pmatrix} = \begin{pmatrix} x_2 \\ x_3 \oplus x_4 \ll 3 \\ x_4 \oplus x_{M_R,4} \ll 3 \\ x_{M_R,4} \\ x_3 \gg 2 \oplus x_6 \oplus x_7 \gg 3 \\ x_4 \gg 2 \oplus x_7 \\ x_0 \\ x_1 \end{pmatrix}$$

Multiplication α_5 will thus require 21 operations in total:

- 1 `XOR`, 1 shift and 1 assignment for $x_{M_R,4}$,
- 5 `XOR`s and 5 shifts,
- 8 byte assignments.

To sum up, **for the 6-lane case**, the update function will require 80 operations (cf. 5-lane case) plus 21 operations for α_5, i.e. 101 operations. After adding $8 \times (6-1) + 1 = 41$ `XOR`s for the extraction function, we reach **142 operations for the whole subtweakey computation**.

7-lane case

Finally, to process seven lanes, multiplications $\alpha_0 = I$, $\alpha_1 = M$, $\alpha_2 = M^2$, $\alpha_3 = M^3$, $\alpha_4 = M_R$, $\alpha_5 = M_R^2$ (already described) and $\alpha_6 = M_R^3$ are needed.

Multiplication α_6 of vector $x = (x_0, x_1, \cdots, x_7)^t$ by matrix M_R^3 corresponds to three successive applications of M_R. Let us denote $M_R^2 \cdot x$ as $x^2_{M_R} = (x_{M_R^2,0}, \cdots, x_{M_R^2,7})$:

$$\begin{pmatrix} x_{M_R^2,0} \\ x_{M_R^2,1} \\ x_{M_R^2,2} \\ x_{M_R^2,3} \\ x_{M_R^2,4} \\ x_{M_R^2,5} \\ x_{M_R^2,6} \\ x_{M_R^2,7} \end{pmatrix} = \begin{pmatrix} x_{M_R,1} \\ x_{M_R,2} \\ x_{M_R,3} \oplus x_{M_R,4} \ll 3 \\ x_{M_R,4} \\ x_{M_R,5} \oplus x_{M_R,6} \gg 3 \\ x_{M_R,3} \gg 2 \oplus x_{M_R,6} \\ x_{M_R,7} \\ x_{M_R,0} \end{pmatrix}$$

$y = M_R^3 \cdot x$ can then be expressed as:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} x_{M_R^2,1} \\ x_{M_R^2,2} \\ x_{M_R^2,3} \oplus x_{M_R^2,4} \ll 3 \\ x_{M_R^2,4} \\ x_{M_R^2,5} \oplus x_{M_R^2,6} \gg 3 \\ x_{M_R^2,3} \gg 2 \oplus x_{M_R^2,6} \\ x_{M_R^2,7} \\ x_{M_R^2,0} \end{pmatrix}$$

Only $x_{M_R^2,4} = x_{M_R,5} \oplus x_{M_R,6} \gg 3 = x_3 \gg 2 \oplus x_6 \oplus x_7 \gg 3$ depends on more than one components of x_{M_R} while contributing to more than one components of y. To implement α_6 using as few operations as necessary, we thus need to:

1. pre-compute $x_{M_R,4}$
2. pre-compute $x_{M_R^2,4}$,
3. compute y as follows:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} x_{M_R^2,1} \\ x_{M_R^2,2} \\ x_{M_R^2,3} \oplus x_{M_R^2,4} \lll 3 \\ x_{M_R^2,4} \\ x_{M_R^2,5} \oplus x_{M_R^2,6} \ggg 3 \\ x_{M_R^2,3} \ggg 2 \oplus x_{M_R^2,6} \\ x_{M_R^2,7} \\ x_{M_R^2,0} \end{pmatrix} = \begin{pmatrix} x_{M_R,2} \\ x_{M_R,3} \oplus x_{M_R,4} \lll 3 \\ x_{M_R,4} \oplus x_{M_R^2,4} \lll 3 \\ x_{M_R^2,4} \\ x_{M_R,5} \oplus x_{M_R,6} \ggg 3 \oplus x_{M_R,7} \ggg 3 \\ x_{M_R,4} \ggg 2 \oplus x_{M_R,7} \\ x_{M_R,0} \\ x_{M_R,1} \end{pmatrix} = \begin{pmatrix} x_3 \oplus x_4 \lll 3 \\ x_4 \oplus x_{M_R,4} \lll 3 \\ x_{M_R,4} \oplus x_{M_R^2,4} \lll 3 \\ x_{M_R^2,4} \\ x_3 \ggg 2 \oplus x_6 \oplus x_7 \ggg 3 \\ x_{M_R,4} \ggg 2 \oplus x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

Overall, **multiplication α_6 will require 28 operations:**

- 1 `XOR`, 1 shift and 1 assignment for $x_{M_R,4}$,
- 2 `XOR`s, 2 shifts and 1 assignment for $x_{M_R^2,4}$,
- 6 `XOR`s and 6 shifts,
- 8 byte assignments.

To sum up, **for the 7-lane case**, the update function will require 101 operations (cf. 6-lane case) plus 28 operations for α_6, i.e. 129 operations. After adding $8 \times (7-1)+1 = 49$ `XOR`s for the extraction function, we reach **178 operations for the whole subtweakey computation**.

4.1.3 Possible Trade-Offs

Another implementation of the tweakey schedule's update function can save some program space at the expense of extra RAM usage and latency. To multiply a lane x by matrix M (resp. M_R) raised to the power of $n > 1$, one can multiply x by M (resp. M_R) n times, instead of using the ad-hoc expressions given in section 4.1.2. This allows the implementer to re-use the code for α_1 (resp. α_4) in order to compute α_2 and α_3 (resp. α_5 and α_6), although computing and storing each byte of every $M^i \cdot x, \forall i \in [1, n)$ requires more cycles and more working memory.

4.1.4 16-bit and 32-bit Platforms

Typical implementations of the Lilliput-TBC tweakey schedule and `OneRoundEGFN` function map each lane and (left and right) parts of Feistel network to a CPU word, resulting in the state of Lilliput-AE represented in 6 to 9 words of 64 bits each depending on the number of lanes.

Specifically, the implementation of Lilliput-AE on a 64-bit CPU can exploit 64-bit wide boolean operations and 64-bit rotations. Thus, the choice of Lilliput-AE favors 64-bit CPUs and yet remains efficient on 32-bit (and smaller) processors.

For implementing the tweakey schedule on a 32-bit CPU, the 64 bits of a lane should be distributed to two 32-bit words.

Implementing `OneRoundEGFN` computation leaves room for optimization on 16-bit and 32-bit processors. For 16-bit processors case, bytes should be considered and concatenated two-by-two ($x_1||x_0$, $x_3||x_2$, and so on) and then `XOR` operations can be extended to 16 bits. For 32-bit processors case, bytes should be considered and concatenated four-by-four ($x_3||x_2||x_1||x_0$, $x_7||x_6||x_5||x_4$, and so on) to get the same kind of benefits for 32-bit `XOR` operations. To ease the schedule of operations, the end of `LinearLayer` (algorithm 7) should be computed before the beginning (algorithm 8) to compute repetitive 16-bit or 32-bit `XOR` operations.

Algorithm 7: Linear layer - second loop

for $i = 0$ **to** 6 **do**
    $x_{14-i} \leftarrow x_{14-i} \oplus x_7$

Algorithm 8: Linear layer - first loop

```
1 for i = 0 to 7 do
2   x15 ← x15 ⊕ xi
```

p	Nb of operations	**Cost wrt. 4-lane case**
4	144	1
5	166	1.15
6	195	1.35
7	231	1.60

Table 4.2: Relative cost of a single round of Lilliput-TBC $\forall p \in [4, 7]$. "Nb of operations" is the sum of the number of operations for `OneRoundEGFN` (53, cf. 4.1.1) and the subtweakey computation (cf. 4.1.2).

4.1.5 Performance Benchmarks Summary

In terms of memory footprint, `OneRoundEGFN` function of Lilliput-TBC can fit easily in the working memory (internal registers) of any considered processor, without requiring any additional RAM register. For example, 8-bit Atmel AVR ATmega128 processors implement 32×8-bit registers, and then, since only 16 internal registers are needed to process the entire internal state, it leaves room for 16 more available registers for intermediate computations. Concerning the tweakey schedule, since computations on each lane are executed separately, and only at most 3 additional registers are needed to compute the more complex operation (α_3), RAM stack consumption is very low.

A first theoretical approximation gives that Lilliput-TBC requires at most (i.e. for the 7-lane case) 72 bytes of RAM for the entire internal state and some working memory.

Table 4.2 compares the relative performance of a single round of each variant of the Lilliput-TBC family.

This subsection also showcases comparisons with other lightweight AEAD algorithms. We chose to compare Lilliput-AE with submissions to the CAESAR [16] competition; in particular, we focused on the final portfolio for use-case 1, which includes Ascon [22] and ACORN [61]. The features of this specific portfolio [9] align with Lilliput-AE's own characteristics:

```
Use Case 1: Lightweight applications (resource constrained environments)
* critical: fits into small hardware area and/or small code for 8-bit CPUs
* desirable: natural ability to protect against side-channel attacks
* desirable: hardware performance, especially energy/bit
* desirable: speed on 8-bit CPUs
* message sizes: usually short (can be under 16 bytes), sometimes longer
```

A customized version of the FELICS framework [24] has been developed to evaluate the code size, RAM consumption and execution time of these block ciphers on three microcontrollers:

- an 8-bit AVR ATmega128,
- a 16-bit TI MSP430,
- a 32-bit Arduino Due with ARM Cortex M3.

Our aim is to compare the performance of these block ciphers in a typical IoT situation: the test scenario thus consists in encrypting a single 16-byte message along with 16-byte associated data. The following compiler options were tested:

- `-O3`, to minimize computation time in order to decrease power consumption,
- `-Os`, to reduce code size and thus optimize for low memory footprint.

The FELICS framework was run on an Ubuntu 16.04 64-bit desktop with 4 3.5 GHz CPUs and 8 GB RAM. The software versions for platform-specific compilers, debuggers and other such utilities correspond to those distributed by Ubuntu, with the exception of software listed in table 4.3.

Platform	**Software**	**Version**	**Origin**
AVR	simavr	`v1.6`	Developer release [46]
	Avrora	`1.7.117-patched`	Cf. FELICS documentation [23]
MSP	MSP430-GCC	`7.3.2.154`	Texas Instruments [42]
	MSPDebug	`v0.25`	Developer release [5]
ARM	J-Link Software	`V6.42f`	SEGGER [31]

Table 4.3: Software versions for the FELICS framework.

The source code for the CAESAR algorithms was adapted from the SUPERCOP [56] toolkit in order to comply with FELICS's requirements. This implied, among other things:

- replacing platform-specific integer types with the exact-width types defined in `stdint.h`,
- isolating encryption code, decryption code, as well as constants held in read-only memory, into distinct files,
- specifying where buffers should be stored (program memory or RAM) and how they should be aligned, using FELICS-specific macro annotations.

In order to provide a fair assessment of each algorithm's performance, we looked for implementations of Ascon and ACORN distributed with SUPERCOP that performed well (i.e. better than the reference version) for each FELICS platform. Table 4.4 sums up which implementations were considered for each platform.

Algorithm	**Platform**	**Implementations**
Ascon	AVR	`ref`
	MSP	`ref`
	ARM	`ref`, `opt32`
	PC	`ref`, `opt64`
ACORN	AVR	`8bitfast`
	MSP	`8bitfast`
	ARM	`opt1`
	PC	`opt1`

Table 4.4: Algorithm implementations for each platform

The `felicsref` implementation of Lilliput-AE used in this benchmark differs from the reference implementation only in trivial ways. More specifically, the reference tweakey schedule code features a loop over an array of function pointers; we manually unrolled this loop and replaced the function pointers with direct calls. This was found to improve performance on every metric, for every platform, for every compilation option, with gains ranging from 1% to 4% for code size, 1.5% to 3% for RAM usage, 10% to 30% for execution time.

Tables 4.5, 4.6, 4.7 and 4.8 give our results for all 128-key algorithms on ATmega128, MSP430, ARM and desktop PC respectively. These results showcase performance for the full encryption process, including key (or tweakey) schedule.

	Version	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
ACORN-128	8bitfast	-O3	3700	263	287991
Ascon-128	ref	-O3	6140	268	191049
Ascon-128a	ref	-O3	6832	300	163320
Lilliput-I-128	felicsref	-O3	7144	519	130702
Lilliput-II-128	felicsref	-O3	6476	496	134645
ACORN-128	8bitfast	-Os	2850	240	335934
Ascon-128	ref	-Os	4322	323	254913
Ascon-128a	ref	-Os	4340	339	216080
Lilliput-I-128	felicsref	-Os	3166	514	189818
Lilliput-II-128	felicsref	-Os	3096	478	231733

Table 4.5: Performance results for 128-bit key algorithms on AVR ATmega128.

	Version	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
ACORN-128	8bitfast	-O3	3276	274	391983
Ascon-128	ref	-O3	8358	290	544075
Ascon-128a	ref	-O3	8620	306	457998
Lilliput-I-128	felicsref	-O3	8194	618	126129
Lilliput-II-128	felicsref	-O3	6162	592	149366
ACORN-128	8bitfast	-Os	2326	218	381698
Ascon-128	ref	-Os	3686	372	567110
Ascon-128a	ref	-Os	3672	382	475176
Lilliput-I-128	felicsref	-Os	2530	538	208848
Lilliput-II-128	felicsref	-Os	2524	508	247450

Table 4.6: Performance results for 128-bit key algorithms on MSP430F1611.

	Version	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
ACORN-128	opt1	-O3	7608	808	56275
Ascon-128	opt32	-O3	18912	268	12790
Ascon-128	ref	-O3	4080	600	32350
Ascon-128a	opt32	-O3	23764	272	11714
Ascon-128a	ref	-O3	4424	608	27683
Lilliput-I-128	felicsref	-O3	6316	748	87002
Lilliput-II-128	felicsref	-O3	5256	724	93825
ACORN-128	opt1	-Os	2364	344	44901
Ascon-128	opt32	-Os	16072	240	10221
Ascon-128	ref	-Os	1426	472	49636
Ascon-128a	opt32	-Os	18996	256	9297
Ascon-128a	ref	-Os	1408	480	41113
Lilliput-I-128	felicsref	-Os	1744	584	184296
Lilliput-II-128	felicsref	-Os	1774	552	218396

Table 4.7: Performance results for 128-bit key algorithms on ARM Cortex-M3.

	Version	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
ACORN-128	opt1	-O3	6122	448	3045
ASCON-128	opt64	-O3	9616	192	1372
ASCON-128	ref	-O3	2236	1984	6775
ASCON-128A	opt64	-O3	11562	200	1169
ASCON-128A	ref	-O3	2102	1984	6421
LILLIPUT-I-128	felicsref	-O3	8414	896	10538
LILLIPUT-II-128	felicsref	-O3	7340	880	12002
ACORN-128	opt1	-Os	2564	392	3816
ASCON-128	opt64	-Os	9074	184	1389
ASCON-128	ref	-Os	1486	448	4046
ASCON-128A	opt64	-Os	10430	180	1316
ASCON-128A	ref	-Os	1466	448	3686
LILLIPUT-I-128	felicsref	-Os	2822	704	18126
LILLIPUT-II-128	felicsref	-Os	2783	688	22073

Table 4.8: Performance results for 128-bit key algorithms on PC.

Finally, tables 4.9, 4.10, 4.11 and 4.12 show the performance of the `felicsref` version of each member of the LILLIPUT-AE family.

	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
LILLIPUT-I-128	-O3	7144	519	130702
LILLIPUT-I-192	-O3	7236	567	163531
LILLIPUT-I-256	-O3	7348	631	211705
LILLIPUT-II-128	-O3	6476	496	134645
LILLIPUT-II-192	-O3	6420	548	195029
LILLIPUT-II-256	-O3	6504	612	253369
LILLIPUT-I-128	-Os	3166	514	189818
LILLIPUT-I-192	-Os	3268	562	230309
LILLIPUT-I-256	-Os	3392	626	290363
LILLIPUT-II-128	-Os	3096	478	231733
LILLIPUT-II-192	-Os	3168	526	281037
LILLIPUT-II-256	-Os	3260	590	354293

Table 4.9: Performance of LILLIPUT-AE on AVR ATmega128.

	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
LILLIPUT-I-128	-O3	8194	618	126129
LILLIPUT-I-192	-O3	8380	666	159828
LILLIPUT-I-256	-O3	8612	732	212328
LILLIPUT-II-128	-O3	6162	592	149366
LILLIPUT-II-192	-O3	6312	640	187066
LILLIPUT-II-256	-O3	6498	704	246254
LILLIPUT-I-128	-Os	2530	538	208848
LILLIPUT-I-192	-Os	2652	586	254871
LILLIPUT-I-256	-Os	2820	652	325317
LILLIPUT-II-128	-Os	2524	508	247450
LILLIPUT-II-192	-Os	2610	556	301102
LILLIPUT-II-256	-Os	2732	620	383498

Table 4.10: Performance of LILLIPUT-AE on MSP430F1611.

	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
Lilliput-I-128	-O3	6316	748	87002
Lilliput-I-192	-O3	6420	796	108311
Lilliput-I-256	-O3	6544	868	143406
Lilliput-II-128	-O3	5256	724	93825
Lilliput-II-192	-O3	5172	772	129692
Lilliput-II-256	-O3	5276	836	168361
Lilliput-I-128	-Os	1744	584	184296
Lilliput-I-192	-Os	1836	632	248497
Lilliput-I-256	-Os	1940	696	295961
Lilliput-II-128	-Os	1774	552	218396
Lilliput-II-192	-Os	1852	600	299974
Lilliput-II-256	-Os	1942	664	358764

Table 4.11: Performance of Lilliput-AE on ARM Cortex-M3.

	CFLAGS	Code size (B)	RAM (B)	Execution time (cycles)
Lilliput-I-128	-O3	8414	896	10538
Lilliput-I-192	-O3	8607	952	13071
Lilliput-I-256	-O3	8831	1008	16776
Lilliput-II-128	-O3	7340	880	12002
Lilliput-II-192	-O3	7504	920	15105
Lilliput-II-256	-O3	7697	992	19694
Lilliput-I-128	-Os	2822	704	18126
Lilliput-I-192	-Os	2967	760	20548
Lilliput-I-256	-Os	3118	816	26418
Lilliput-II-128	-Os	2783	688	22073
Lilliput-II-192	-Os	2882	728	26151
Lilliput-II-256	-Os	3018	800	33894

Table 4.12: Performance of Lilliput-AE on PC.

4.2 Hardware Implementations

4.2.1 Theoretical Results on ASIC

In this section, we provide theoretical hardware implementation results on ASIC (Application-Specific Integrated Circuit) in terms of GEs. One GE is the area of a 2-input NAND gate in the considered CMOS technology. It allows to get normalized area and then ease comparisons between different implementations that use the same CMOS technology.

We provide here the global logic gates count for each lanes case, and translate it to the total number of GEs in a given CMOS technology. That respectively allows the reader to easily get estimations for other CMOS technologies and get real implementation numbers. The CMOS technology used here is UMCL18G212T3 (CMOS 180 nm technology). In this technology, area of respectively XOR, NOT, AND gates, and flip-flops are 2.67, 0.67, 1.33 and 5.33 GEs. We use non-scan flip-flops for registers in this estimation. Moreover, control logic (e.g., multiplexers, finite state machine) is not taken into account, which can underestimate in the end the real practical results after Place-and-Route process. We also give a relative performance metric, which gives an estimation of the percentage of circuit area increase (considering the total number of GEs) for each lanes case, with the 4-lane case considered as a reference. We can estimate that one Lilliput-TBC S-box is equivalent to the total size of 12 AND, 26 XOR and 1 NOT gates, and so: $12 \times 1.33 + 26 \times 2.67 + 1 \times 0.67 = 15.96 + 69.42 + 0.67 \approx 86$ GEs.

Nb. Lanes	Registers	Round Function	Tweakey Schedule	Total	Relative Perf.
4	384	8 S-boxes + 29×8 XORs	176 XORs	4057 GEs	1
5	448	8 S-boxes + 29×8 XORs	200 XORs	4230 GEs	1.04
6	512	8 S-boxes + 29×8 XORs	256 XORs	4721 GEs	1.16
7	576	8 S-boxes + 29×8 XORs	354 XORs	4983 GEs	1.22

We can compare these results with other hardware implementations of cryptographic standards. One of the most compact implementations of AES is the "Atomic v2" version [2]: it is very lightweight and smaller than our Lilliput-TBC hardware implementations (only 2060 GEs) but processes data with a big latency (246 cycles) and then a low throughput (88.4 Mbps). One of the most compact implementation of SHA-3 (with 1088-bit block size) occupies 5522 GEs (which is bigger than any version of Lilliput-TBC), and provides a very poor throughput (44.3 kbits) [37].

An argument against tunable parameters in a standard is that it makes implementations more expensive, as they usually have to support all parameter values to fully implement the standard. However, for Lilliput-AE, this can be mitigated by only implementing the hardware needed for computing the M and M_R functions, and iterate on them to compute the needed remaining multiplications. This version will allow to save some logic gates, but at the expense of a decreased throughput.

For the FPGA implementation particular case, tables M_1, M_2, M_3, M_4 and the S-box S can be put in dedicated block RAMs of the used FPGA.

The high parallelization level of the nonce-respecting and the nonce-misuse resistant modes allows implementing in hardware many instances of E_K running in parallel and then getting high throughput, especially on dedicated ASICs.

4.2.2 VHDL Results

This subsection showcases performance results for iterated versions of all variants of the Lilliput-TBC tweakable block cipher. These results were produced using version 14.4 of the ISE Design Suite on a Virtex-6 XC6VLX75T device, with two optimization settings: "area reduction" and "timing performance".

Tables 4.13 and 4.14 provide results for encryption with implementations optimized for circuit area and execution time, respectively. Similarly, tables 4.15 and 4.16 provide the same data for the decryption process, tables 4.17 and 4.18 for both operations combined.

Finally, tables 4.19 and 4.20 show how Lilliput-TBC compares to Ascon-128a when optimized for circuit area and execution time, respectively. We used the iterated implementation of Ascon-128a described in [25].

Table 4.13: Results for Lilliput-TBC encryption, optimized for area reduction.

Lilliput-TBC	I-128	I-192	I-256	II-128	II-192	II-256
LUTs	1,009	1,175	1,166	1,393	1,331	1,552
slices	285	326	337	388	368	429
registers	876	1,008	1,009	1,137	1,137	1,269
flip-flop pairs	1,030	1,175	1,201	1,415	1,349	1,582
unused flip-flops	452	739	455	507	510	498
unused LUTs	21	14	35	22	18	30
fully used	557	422	711	886	821	1,054
REQ SYN TCLK	4.3	4.7	4.8	4.7	5	5
SYN TCLK (ns)	4.29	4.534	4.761	4.694	4.95	4.94
IMP FREQ (MHz)	233.1	220.556	210.04	213.038	201.979	202.429

Table 4.14: Results for LILLIPUT-TBC encryption, optimized for timing performance.

LILLIPUT-TBC	II-128	I-128	II-192	I-192	II-256	I-256
LUTs	1,096	1,275	1,277	1,481	1,417	1,771
slices	323	348	373	460	401	576
registers	910	1,043	1,044	1,174	1,174	1,307
flip-flop pairs	1,102	1,276	1,282	1,482	1,418	1,781
unused flip-flops	463	462	470	518	518	605
unused LUTs	6	1	5	1	1	10
fully used	633	813	807	963	899	1,166
REQ SYN TCLK	3.3	3.3	3.3	3.8	3.8	3.8
SYN TCLK (ns)	3.147	3.145	3.149	3.712	3.648	3.647
IMP FREQ (MHz)	317.763	317.965	317.561	269.397	274.123	274.198

Table 4.15: Results for LILLIPUT-TBC decryption, optimized for area reduction.

LILLIPUT-TBC	I-128	I-192	I-256	II-128	II-192	II-256
LUTs	1,324	1,548	2,019	1,068	1,279	1,503
slices	393	433	543	301	377	446
registers	1,014	1,142	1,274	882	1,014	1,142
flip-flop pairs	1,343	1,561	2,019	1,082	1,298	1,524
unused flip-flops	513	547	856	468	470	561
unused LUTs	19	13	0	14	19	21
fully used	811	1,001	1,163	600	809	942
Imp Clk (ns)	5.485	5.493	5.496	5.45	5.588	6.433
Imp Freq (MHz)	182.315	182.05	181.951	183.486	178.955	155.448

Table 4.16: Results for LILLIPUT-TBC decryption, optimized for timing performance.

LILLIPUT-TBC	I-128	I-192	I-256	II-128	II-192	II-256
LUTs	1,381	1,540	1,870	1,054	1,429	1,487
slices	422	448	564	406	457	452
registers	1,193	1,197	1,476	1,145	1,193	1,198
flip-flop pairs	1,453	1,590	1,946	1,259	1,516	1,531
unused flip-flops	531	526	618	513	582	533
unused LUTs	72	50	76	114	87	44
fully used	850	1,014	1,252	632	847	954
Imp Clk (ns)	4.93	5.39	5.269	5.089	5.1	4.942
Imp Freq (MHz)	202.84	185.529	189.789	196.502	196.078	202.347

Table 4.17: Results for LILLIPUT-TBC encryption and decryption, optimized for area reduction.

LILLIPUT-TBC	I-128	I-192	I-256	II-128	II-192	II-256
LUTs	1,506	1,634	1,894	1,088	1,507	1,716
slices	391	468	521	309	395	455
registers	1,017	1,145	1,277	885	1,017	1,145
flip-flop pairs	1,506	1,637	1,895	1,094	1,507	1,716
unused flip-flops	626	585	722	441	624	705
unused LUTs	0	3	1	6	0	0
fully used	880	1,049	1,172	647	883	1,011
Imp Clk (ns)	5.398	5.463	5.653	5.398	5.647	5.535
Imp Freq (MHz)	185.254	183.05	176.89	185.254	177.085	180.668

Table 4.18: Results for Lilliput-TBC encryption and decryption, optimized for timing performance.

Lilliput-TBC	I-128	I-192	I-256	II-128	II-192	II-256
LUTs	1,450	1,659	1,954	1,267	1,431	1,589
slices	482	542	574	427	416	446
registers	1,207	1,283	1,454	1,077	1,211	1,283
flip-flop pairs	1,564	1,758	2,023	1,369	1,482	1,619
unused flip-flops	543	605	703	483	461	540
unused LUTs	114	99	69	102	51	30
fully used	907	1,054	1,251	784	970	1,049
Imp Clk (ns)	5.402	5.186	5.665	5.285	5.256	5.061
Imp Freq (MHz)	185.117	192.827	176.523	189.215	190.259	197.589

Table 4.19: Comparison of implementations optimized for area reduction.

	Ascon-128a	Lilliput-TBC-I-128	Lilliput-TBC-II-128
LUTs	1,435	1,508	1,063
slices	383	399	305
registers	963	1,017	885
flip-flop pairs	1,440	1,508	1,070
unused flip-flops	721	626	412
unused LUTs	5	0	7
fully used	714	882	651
SYN TCLK (ns)	2.412	5.491	5.398
IMP FREQ (MHz)	414.594	182.116	185.254

Table 4.20: Comparison of implementations optimized for timing performance.

	Ascon-128a	Lilliput-TBC-I-128	Lilliput-TBC-II-128
LUTs	1,461	1,450	1,267
slices	548	482	427
registers	1,353	1,207	1,077
flip-flop pairs	1,740	1,564	1,369
unused flip-flops	637	543	483
unused LUTs	279	114	102
fully used	824	907	784
SYN TCLK (ns)	3.227	5.402	5.285
IMP FREQ (MHz)	309.885	185.117	189.215

4.3 Threshold Implementations

This section aims at giving the reader some insight into first order TIs of Lilliput-AE.

4.3.1 The S-box

The quadratic functions

As stated in Section 3.2.3, the 8-bit S-box has been chosen with TIs in mind as it is built from three inner 4-bit S-boxes, each directly decomposable into quadratic permutations. Therefore, a first order TI can be achieved using only three shares. The following algorithms describe, for each quadratic permutation F, G and Q, a function f that computes an output share $\langle x, y, z, t\rangle$ for two input shares $\langle a_0, b_0, c_0, d_0\rangle$ and $\langle a_1, b_1, c_1, d_1\rangle$.

Algorithm 9: $f_F(\langle a_0, b_0, c_0, d_0\rangle, \langle a_1, b_1, c_1, d_1\rangle) = \langle x, y, z, t\rangle$

$x \leftarrow (a_0 \oplus c_0)(b_0 \oplus d_0) \oplus (a_0 \oplus c_0)(b_1 \oplus d_1) \oplus (a_1 \oplus c_1)(b_0 \oplus d_0)$
$y \leftarrow a_0 d_0 \oplus a_0 d_1 \oplus a_1 d_0$
$z \leftarrow b_1 \oplus d_1$
$t \leftarrow (a_0 \oplus b_0 \oplus d_0)(a_0 \oplus b_0 \oplus c_0) \oplus (a_0 \oplus b_0 \oplus d_0)(a_1 \oplus b_1 \oplus c_1) \oplus (a_1 \oplus b_1 \oplus d_1)(a_0 \oplus b_0 \oplus c_0)$

Algorithm 10: $f_G(\langle a_0, b_0, c_0, d_0\rangle, \langle a_1, b_1, c_1, d_1\rangle) = \langle x, y, z, t\rangle$

$x \leftarrow a_1$
$y \leftarrow b_1$
$z \leftarrow c_1$
$t \leftarrow b_0 c_0 \oplus b_0 c_1 \oplus b_1 c_0 \oplus d_1$

Algorithm 11: $f_Q(\langle a_0, b_0, c_0, d_0\rangle, \langle a_1, b_1, c_1, d_1\rangle) = \langle x, y, z, t\rangle$

$x \leftarrow c_0 d_0 \oplus c_0 d_1 \oplus c_1 d_0 \oplus b_1$
$y \leftarrow d_1$
$z \leftarrow a_0 d_0 \oplus a_0 d_1 \oplus a_1 d_0 \oplus c_1$
$t \leftarrow a_1$

Therefore, for each quadratic function $A \in F, G, Q$, TI with three shares is achived by computing

$$\begin{aligned} A(\langle a_0, b_0, c_0, d_0\rangle, \langle a_1, b_1, c_1, d_1\rangle, \langle a_2, b_2, c_2, d_2\rangle) = & f_A(\langle a_1, b_1, c_1, d_1\rangle, \langle a_2, b_2, c_2, d_2\rangle), \\ & f_A(\langle a_2, b_2, c_2, d_2\rangle, \langle a_0, b_0, c_0, d_0\rangle), \\ & f_A(\langle a_0, b_0, c_0, d_0\rangle, \langle a_1, b_1, c_1, d_1\rangle)\,. \end{aligned} \tag{4.1}$$

Contrary to Q and G, the output sharing of F is not uniform but it does not matter as these functions are used in a Feistel network. Therefore, there is no need for re-masking and a threshold implementation of the 8-bit S-box can be built upon the algorithms described above. Note that the inner 4-bit S-box S_4^3 requires an additionnal `NOT` instruction: it only has to be applied to one of the three shares (i.e., $\neg x = \neg x_0 \oplus x_1 \oplus x_2$).

Software implementation using Look-Up Tables

In order to improve the performance of software implementations, it is possible to use look-up tables for the quadratic functions as done in [54]. To do so, one can compute three 8-bit to 4-bit look-up tables from f_F, f_G and f_Q noted T_F, T_G and T_Q, respectively. Because $\bar{S}_4^2$ requires a bitwise permutation $P =$ `028a46ce139b57df` between the two quadratics, an additionnal 4-bit to 4-bit look-up table can be used.

However, as a_0 and d_0 do not interfere in the computation of $f_G(\langle a_0, b_0, c_0, d_0\rangle, \langle a_1, b_1, c_1, d_1\rangle)$, it is possible to divide the size of T_G by four (i.e., from 256 to 64 bytes) at the cost of two bitwise operations at each table look-up. In the same way, b_0 does not interfere in the computation of $f_Q(\langle a_0, b_0, c_0, d_0\rangle, \langle a_1, b_1, c_1, d_1\rangle)$ and the size of T_Q can be reduced by half. In the rest of this section, we use these tricks in order to minimize the memory space required to store the look-up tables. The three resulting look-up tables are given below.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	0	2	0	2	2	0	2	0	0	2	0	2	2	0	2	0
1	0	2	9	b	3	1	a	8	d	f	4	6	e	c	7	5
2	0	b	0	b	b	0	b	0	1	a	1	a	a	1	a	1
3	9	2	0	b	3	8	a	1	5	e	c	7	f	4	6	d
4	1	2	8	b	3	0	a	9	9	a	0	3	b	8	2	1
5	0	3	0	3	3	0	3	0	5	6	5	6	6	5	6	5
6	8	2	1	b	3	9	a	0	1	b	8	2	a	0	3	9
7	0	a	0	a	a	0	a	0	4	e	4	e	e	4	e	4
8	1	e	0	f	b	4	a	5	1	e	0	f	b	4	a	5
9	c	3	4	b	7	8	f	0	1	e	9	6	a	5	2	d
a	0	6	1	7	3	5	2	4	1	7	0	6	2	4	3	5
b	4	2	c	a	6	0	e	8	8	e	0	6	a	c	2	4
c	8	6	0	e	2	c	a	4	0	e	8	6	a	4	2	c
d	4	a	5	b	f	1	e	0	1	f	0	e	a	4	b	5
e	0	7	8	f	3	4	b	c	9	e	1	6	a	d	2	5
f	5	2	4	3	7	0	6	1	1	6	0	7	3	4	2	5

Table 4.21: $T_F[x][y] = f_F(x, y)$

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
1	0	1	2	3	5	4	7	6	8	9	a	b	d	c	f	e
2	0	1	3	2	4	5	7	6	8	9	b	a	c	d	f	e
3	1	0	2	3	4	5	7	6	9	8	a	b	c	d	f	e

Table 4.22: $T_G[x][y] = f_G(x \ll 1, y)$

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	0	4	2	6	8	c	a	e	1	5	3	7	9	d	b	f
1	0	4	a	e	8	c	2	6	3	7	9	d	b	f	1	5
2	0	c	2	e	8	4	a	6	1	d	3	f	9	5	b	7
3	8	4	2	e	0	c	a	6	b	7	1	d	3	f	9	5
4	0	6	2	4	8	e	a	c	1	7	3	5	9	f	b	d
5	2	4	8	e	a	c	0	6	1	7	b	d	9	f	3	5
6	0	e	2	c	8	6	a	4	1	f	3	d	9	7	b	5
7	a	4	0	e	2	c	8	6	9	7	3	d	1	f	b	5

Table 4.23: $T_Q[x][y] = f_Q(x + 4, y)$

In this way, the memory space required to store all the look-up tables equals $|T_F| + |T_G| + |T_Q| + |P| = 256 + 64 + 128 + 16 = 464$ bytes. Finally, the output shares of the 8-bit S-box can be computed by running the Feistel network step by step as detailed by Algorithm 12.

Algorithm 12: $S'(s_0, s_1, s_2) = s'_0, s'_1, s'_2$ with look-up tables T_F, T_G, T_Q and P

/* *Decompose 8-bit shares into 4-bit shares* */
for $i = 0$ **to** 2 **do**
    $\bar{s}_i \leftarrow s_i \gg 4$
    $\underline{s}_i \leftarrow \texttt{AND}(s_i, 15)$
end

/* *First 4-bit S-box* */
$t_0 \leftarrow T_G[\texttt{AND}(\underline{s}_1, 7) \gg 1][\underline{s}_2]$
$t_1 \leftarrow T_G[\texttt{AND}(\underline{s}_2, 7) \gg 1][\underline{s}_0]$
$t_2 \leftarrow T_G[\texttt{AND}(\underline{s}_0, 7) \gg 1][\underline{s}_1]$
$\bar{s}_0 \leftarrow \bar{s}_0 \oplus T_F[t_1][t_2]$
$\bar{s}_1 \leftarrow \bar{s}_1 \oplus T_F[t_2][t_0]$
$\bar{s}_2 \leftarrow \bar{s}_2 \oplus T_F[t_0][t_1]$

/* *Second 4-bit S-box* */
$t_0 \leftarrow P\,[T_Q[\texttt{AND}(\bar{s}_1, 3) \oplus (\texttt{AND}(\bar{s}_1, 8) \gg 1)][\bar{s}_2]]$
$t_1 \leftarrow P\,[T_Q[\texttt{AND}(\bar{s}_2, 3) \oplus (\texttt{AND}(\bar{s}_2, 8) \gg 1)][\bar{s}_0]]$
$t_2 \leftarrow P\,[T_Q[\texttt{AND}(\bar{s}_0, 3) \oplus (\texttt{AND}(\bar{s}_9, 8) \gg 1)][\bar{s}_1]]$
$\underline{s}_0 \leftarrow \underline{s}_0 \oplus T_Q[\texttt{AND}(t_1, 3) \oplus (\texttt{AND}(t_1, 8) \gg 1)][t_2]$
$\underline{s}_1 \leftarrow \underline{s}_1 \oplus T_Q[\texttt{AND}(t_2, 3) \oplus (\texttt{AND}(t_2, 8) \gg 1)][t_0]$
$\underline{s}_2 \leftarrow \underline{s}_2 \oplus T_Q[\texttt{AND}(t_0, 3) \oplus (\texttt{AND}(t_0, 8) \gg 1)][t_1]$

/* *Third 4-bit S-box* */
$t_0 \leftarrow T_G[\texttt{AND}(\underline{s}_1, 7) \gg 1][\underline{s}_2] \oplus 1$
$t_1 \leftarrow T_G[\texttt{AND}(\underline{s}_2, 7) \gg 1][\underline{s}_0]$
$t_2 \leftarrow T_G[\texttt{AND}(\underline{s}_0, 7) \gg 1][\underline{s}_1]$
$\bar{s}_0 \leftarrow \bar{s}_0 \oplus T_F[t_1][t_2]$
$\bar{s}_1 \leftarrow \bar{s}_1 \oplus T_F[t_2][t_0]$
$\bar{s}_2 \leftarrow \bar{s}_2 \oplus T_F[t_0][t_1]$

/* *Build 8-bit output shares from 4-bit shares* */
for $i = 0$ **to** 2 **do**
    $s'_i \leftarrow (\bar{s}_i \ll 4) \oplus \underline{s}_i$
end

4.3.2 Application to the Entire Algorithm

The tweakey schedule

Because the key is manipulated along with the tweak during the tweakey schedule, this step must be protected to prevent a side-channel attack. To do so, one can share the tweak and the key into two shares. There is no difficulty to apply TI to the tweakey schedule as it operates in a linear fashion. As a result, the tweakey schedule produces subtweakeys splitted in two shares RTK_0^i and RTK_1^i. In order to limit the amount of randomness to generate, it is possible to share the key only. However, note that non-sharing the tweak implies that a profiling attack against the tweakey schedule would allow to deduce some information on the power consumption model of the device.

The EGFN round function

A way of applying TI to the round function is to share the input block into three shares which are processed during the entire round function. More precisely, if $X_{i,j}$ refers to the i^{th} byte of the j^{th} share of X, then a TI of F_i at round r consists in $F'_i = S'(X_{i,0} \oplus RTK^r_{i,0},\ X_{i,1} \oplus RTK^r_{i,1},\ X_{i,2})$ where S' refers to the Algorithm 12. Because the remaining steps of the round function are linear, it is sufficient to apply it on each share independently.

4.3.3 Performance Impact

We implemented the thresholding scheme described in this section, using lookup tables for the S-box, and compared its performance with our `felicsref` implementation, in the conditions described in section 4.1.5 with the compiler option `-O3`. Table 4.24 shows the impact for each metric on each platform.

Note that the `threshold` implementation used in this benchmark does not include a random number generator; these results therefore do not account for the overhead induced by share initialization.

Platform	Member	$\frac{ROM_{threshold}}{ROM_{felicsref}}$	$\frac{RAM_{threshold}}{RAM_{felicsref}}$	$\frac{cycles_{threshold}}{cycles_{felicsref}}$
AVR	Lilliput-I-128	2.28	1.78	5.06
	Lilliput-I-192	2.27	1.79	4.78
	Lilliput-I-256	2.28	1.80	4.52
	Lilliput-II-128	2.41	1.81	6.24
	Lilliput-II-192	2.43	1.80	5.23
	Lilliput-II-256	2.45	1.81	4.92
MSP	Lilliput-I-128	1.82	1.75	4.38
	Lilliput-I-192	1.82	1.76	4.16
	Lilliput-I-256	1.84	1.77	3.93
	Lilliput-II-128	2.09	1.77	4.80
	Lilliput-II-192	2.09	1.78	4.58
	Lilliput-II-256	2.11	1.79	4.34
ARM	Lilliput-I-128	1.90	1.88	4.68
	Lilliput-I-192	1.90	1.88	4.34
	Lilliput-I-256	1.91	1.86	4.05
	Lilliput-II-128	2.09	1.90	5.54
	Lilliput-II-192	2.11	1.90	4.73
	Lilliput-II-256	2.13	1.89	4.53
PC	Lilliput-I-128	1.68	1.76	4.42
	Lilliput-I-192	1.68	1.75	4.22
	Lilliput-I-256	1.68	1.77	4.03
	Lilliput-II-128	1.78	1.75	5.04
	Lilliput-II-192	1.77	1.77	4.62
	Lilliput-II-256	1.77	1.77	4.45

Table 4.24: Performance impact of the thresholding scheme.

4.4 Future Works

This chapter provided first results and estimations of the performance of software and hardware implementations of Lilliput-AE. In 2019, the co-authors will publish:

- Optimized software implementations of Lilliput-AE on IoT platforms,
- Side-channel protected implementations of Lilliput-AE with performance benchmark,
- Optimized hardware implementations of Lilliput-AE (e.g., serial implementations).

All this work will be published on the PACLIDO projet website: `https://paclido.fr/lilliput-ae/`.

Chapter 5

Acknowledgments

Without the devotion and hard work of Antoine Martinache, Jean-Baptiste Serrou-Soares and Gaëtan Leplus (Airbus CyberSecurity), this proposal would have not been submitted on quality and on time. The co-authors sincerely thank them.

Moreover, this work has been partly funded by the French Direction Générale des Entreprises (DGE) under grant FUI 23 PACLIDO (Protocoles et Algorithmes Cryptographiques Légers pour l'Internet Des Objets).

Bibliography

[1] Francois Arnault, Thierry P. Berger, Marine Minier, and Benjamin Pousse. Revisiting LFSRs for Cryptographic Applications. *IEEE Trans. on Info. Theory*, 57(12):8095–8113, 2011.

[2] S. Banik, A. Bogdanov, and F. Regazzoni. Atomic-AES v2.0. Cryptology ePrint Archive, Report 2016/1005, 2016.

[3] Paulo S. L. M. Barreto, Vincent Rijmen, Scopus Tecnologia S. A, and Cryptomathic Nv. The Whirlpool Hashing Function. In *First open NESSIE Workshop*, 2000.

[4] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404, 2013.

[5] Daniel Beer. mspdebug. `https://github.com/dlbeer/mspdebug`, 2019. Accessed: 2019-03-07.

[6] Thierry P. Berger, Julien Francq, Marine Minier, and Gaël Thomas. Extended generalized feistel networks using matrix representation to propose a new lightweight block cipher: Lilliput. *IEEE Trans. Computers*, 65(7):2074–2089, 2016.

[7] Thierry P. Berger, Marine Minier, and Benjamin Pousse. Software Oriented Stream Ciphers Based upon FCSRs in Diversified Mode. In *Progress in Cryptology - INDOCRYPT 2009*, volume 5922 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2009.

[8] Thierry P. Berger, Marine Minier, and Gaël Thomas. Extended Generalized Feistel Networks using Matrix Representation. In *Selected Areas in Cryptography - SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 289–305. Springer, 2013.

[9] Daniel J. Bernstein. CAESAR use cases. `https://groups.google.com/forum/#!msg/crypto-competitions/DLv193SPSDc/4CeHPvIoBgAJ`, 2016.

[10] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In *Advances in Cryptology - EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 12–23. Springer, 1999.

[11] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. Threshold implementations of all 3x3 and 4x4 s-boxes. In *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2012.

[12] Alex Biryukov and David Wagner. Slide Attacks. In *Fast Software Encryption - FSE '99*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 1999.

[13] Andrey Bogdanov, Huizheng Geng, Meiqin Wang, Long Wen, and Baudoin Collard. Zero-correlation linear cryptanalysis with FFT and improved attacks on ISO standards camellia and CLEFIA. In *Selected Areas in Cryptography - SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2013.

[14] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matt J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[15] Erik Boss, Vincent Grosso, Tim Güneysu, Gregor Leander, Amir Moradi, and Tobias Schneider. Strong 8-bit sboxes with efficient masking in hardware extended version. *Journal of Cryptographic Engineering*, 7(2):149–165, Jun 2017.

[16] Crypto competitions: CAESAR submissions. `https://competitions.cr.yp.to/caesar-submissions.html`. Accessed: 2019-02-21.

[17] Anne Canteaut, Sébastien Duval, and Gaëtan Leurent. Construction of Lightweight S-Boxes Using Feistel and MISTY Structures. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography – SAC 2015*, pages 373–393, Cham, 2016. Springer International Publishing.

[18] Carlos Cid, Tao Huang, Thomas Peyrin, Yu Sasaki, and Ling Song. A security analysis of deoxys and its internal tweakable block ciphers. *IACR Transactions on Symmetric Cryptology*, 2017(3):73–107, Sep. 2017.

[19] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher square. In *Fast Software Encryption, 4th International Workshop, FSE '97*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.

[20] Hüseyin Demirci and Ali Aydin Selçuk. A meet-in-the-middle attack on 8-round AES. In *Fast Software Encryption - FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 116–126. Springer, 2008.

[21] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of Lightweight Block Ciphers for the Internet of Things. *Journal of Cryptographic Engineering*, pages 1–20, 2018.

[22] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to the CAESAR competition: `https://competitions.cr.yp.to/round3/asconv12.pdf`, 2016.

[23] CryptoLUX > FELICS Avrora patch. `https://www.cryptolux.org/index.php/FELICS_Avrora_patch`, 2019. Accessed: 2019-03-07.

[24] Fair Evaluation of LIghtweight Cryptographic Systems. `https://www.cryptolux.org/index.php/FELICS`. Accessed: 2019-02-14.

[25] Michael Fivez. Energy Efficient Hardware Implementations of CAESAR Submissions. Master's thesis, KU Leuven, 2016. Ingrid Verbauwhede (promotor).

[26] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, Kerem Varıcı, François Durvaux, Lubos Gaspar, and Stéphanie Kerckhof. SCREAM v3, August 2015. Submission to the CAESAR competition.

[27] Viet T. Hoang and Phillip Rogaway. On Generalized Feistel Networks. In *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 613–630. Springer, 2010.

[28] Akiko Inoue, Tetsu Iwata, Kazuhiko Minematsu, and Bertram Poettering. Cryptanalysis of ocb2: Attacks on authenticity and confidentiality. Cryptology ePrint Archive, Report 2019/311, 2019. `https://eprint.iacr.org/2019/311`.

[29] Akiko Inoue and Kazuhiko Minematsu. Cryptanalysis of OCB2. *IACR Cryptology ePrint Archive*, 2018:1040, 2018.

[30] Tetsu Iwata. Plaintext recovery attack of OCB2. *IACR Cryptology ePrint Archive*, 2018:1090, 2018.

[31] J-Link Software and Documentation Pack. `https://www.segger.com/downloads/jlink/#J-LinkSoftwareAndDocumentationPack`, 2019. Accessed: 2019-02-26.

[32] Jérémy Jean, Ivica Nikolic, and Thomas Peyrin. Tweaks and keys for block ciphers: The TWEAKEY framework. In *Advances in Cryptology - ASIACRYPT 2014 - Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 2014.

[33] Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. Deoxys v1. 41. *Submitted to CAESAR*, 2016.

[34] Pascal Junod and Serge Vaudenay. FOX : A new family of block ciphers. In *Selected Areas in Cryptography, 11th International Workshop, SAC 2004*, volume 3357 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2004.

[35] Charanjit S. Jutla. Encryption modes with almost free message integrity. In Birgit Pfitzmann, editor, *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, volume 2045 of *Lecture Notes in Computer Science*, pages 529–544. Springer, 2001.

[36] Pierre Karpman and Benjamin Grégoire. The littlun s-box and the fly block cipher. 2016. `https://pdfs.semanticscholar.org/5487/ccb5b874c1e56f1b8468eef2def91de42d36.pdf`.

[37] E.B. Kavun and T. Yalcin. A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications. In *Radio Frequency Identification: Security and Privacy Issues - RFIDSec*, volume 6370 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2010.

[38] Ted Krovetz and Phillip Rogaway. The software performance of authenticated-encryption modes. In *Fast Software Encryption - 18th International Workshop, FSE 2011*, volume 6733 of *Lecture Notes in Computer Science*, pages 306–327. Springer, 2011.

[39] Ted Krovetz and Phillip Rogaway. The software performance of authenticated-encryption modes. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 306–327. Springer, 2011.

[40] Nicolas Marrière, Valérie Nachef, and Emmanuel Volte. Differential attacks on reduced round LILLIPUT. In *Information Security and Privacy - 23rd Australasian Conference, ACISP 2018*, volume 10946 of *Lecture Notes in Computer Science*, pages 188–206. Springer, 2018.

[41] F Moazami, AR Mehrdad, and Hadi Soleimany. Impossible differential cryptanalysis on deoxys-bc-256. *The ISC International Journal of Information Security*, 10(2):93–105, 2018.

[42] MSP430-GCC-OPENSOURCE GCC - Open Source Compiler fro MSP Microcontrollers. `http://www.ti.com/tool/msp430-gcc-opensource`, 2019. Accessed: 2019-03-07.

[43] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *Journal of Cryptology*, 24(2):292–321, Apr 2011.

[44] Thomas Peyrin and Yannick Seurin. Counter-in-tweak: Authenticated encryption modes for tweakable block ciphers. In *Advances in Cryptology - CRYPTO 2016 - Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 33–63. Springer, 2016.

[45] Bertram Poettering. Breaking the confidentiality of OCB2. *IACR Cryptology ePrint Archive*, 2018:1087, 2018.

[46] Michel Pollet. simavr. `https://github.com/buserror/simavr`, 2019. Accessed: 2019-03-07.

[47] Matthieu Rivain. Differential Fault Analysis on DES Middle Rounds. In *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 457–469. Springer, 2009.

[48] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[49] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: a block-cipher mode of operation for efficient authenticated encryption. In Michael K. Reiter and Pierangela Samarati, editors, *CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, November 6-8, 2001.*, pages 196–205. ACM, 2001.

[50] Markku-Juhani O. Saarinen. Cryptographic Analysis of All 4 x 4 - Bit S-Boxes. Cryptology ePrint Archive, Report 2011/218.

[51] Yu Sasaki and Yosuke Todo. New differential bounds and division property of lilliput: Block cipher with extended generalized feistel network. In *Selected Areas in Cryptography - SAC 2016*, volume 10532 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2016.

[52] Yu Sasaki and Yosuke Todo. New impossible differential search tool from design and cryptanalysis aspects - revealing structural properties of several ciphers. In *Advances in Cryptology - EUROCRYPT 2017 - Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 185–215. Springer, 2017.

[53] Yu Sasaki and Yosuke Todo. Tight bounds of differentially and linearly active s-boxes and division property of lilliput. *IEEE Trans. Computers*, 67(5):717–732, 2018.

[54] Pascal Sasdrich, René Bock, and Amir Moradi. Threshold Implementation in Software - Case Study of PRESENT. Cryptology ePrint Archive, Report 2018/189, 2018. `https://eprint.iacr.org/2018/189`.

[55] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher CLEFIA (extended abstract). In *Fast Software Encryption, 14th International Workshop, FSE 2007*, volume 4593 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2007.

[56] SUPERCOP. `https://bench.cr.yp.to/supercop.html`. Accessed: 2019-02-21.

[57] Tomoyasu Suzaki and Kazuhiko Minematsu. Improving the Generalized Feistel. In *Fast Software Encryption - FSE 2010*, volume 6147 of *Lecture Notes in Computer Science*, pages 19–39. Springer, 2010.

[58] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. TWINE : A Lightweight Block Cipher for Multiple Platforms. In *Selected Areas in Cryptography - SAC 2012*, volume 7707 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2012.

[59] Yosuke Todo. Structural evaluation by generalized integral property. In *Advances in Cryptology - EUROCRYPT 2015 - Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 287–314. Springer, 2015.

[60] Markus Ullrich, Christophe Decannière, Sebastiaan Indesteege, Ozgül Küçük, Nicky Mouha, and Bart Preneel. Finding Optimal Bitsliced Implementations of 4x4-bit S-boxes. Feb 2011.

[61] Hongjun Wu. Acorn: a lightweight authenticated cipher (v3). *Candidate for the CAESAR Competition. See also* `https://competitions.cr.yp.to/round3/acornv3.pdf`, 2016.

[62] Wenling Wu and Lei Zhang. LBlock: A Lightweight Block Cipher. In *Applied Cryptography and Network Security - ACNS 2011*, volume 6715 of *Lecture Notes in Computer Science*, pages 327–344, 2011.

[63] Rui Zong, Xiaoyang Dong, and Xiaoyun Wang. Related-tweakey impossible differential attack on reduced-round deoxys-bc-256. *Science China Information Sciences*, 62(3):32102, 2019.